



An Introduction to DeKlarit

DeKlarit is a Rapid Application Development tool designed to let developers build Business Frameworks with an instant increase in productivity while greatly simplifying change management.

DeKlarit is integrated into Visual Studio and is designed to be a perfect match for the .NET developer.

Why DeKlarit?

- An automatic way of building complex applications incrementally with efficient change management.
- A Rapid Application Development tool for the business rules and data access layers.
- No need for expertise in database design or building middle tier components. It allows you to focus on the rest of the application.
- Support for Access, DB/2, DB/2 for iSeries, MySQL, Oracle, SQL Server, and SQL Server 2005 Mobile Edition.
- Projects can be generated in C# or Visual Basic .NET.

The objective of this white paper is to provide information regarding this new and exciting technology. It starts with a quick review of how software has been traditionally developed and then explains how DeKlarit can help you build higher quality applications.

1. Software development methodologies

For a long time, traditional software development methodologies have supported the concept that the application's design should be completed before starting the programming phase. These methodologies were based on the assumption that the cost of change in the design phase is much lower than in the programming phase.

However, these methodologies were not applicable to environments that had to adapt to rapid changes, so clearly a different way to develop 'agile' software was needed.

In recent years the industry has come to terms with this reality, and right now there is consensus in that software should be developed incrementally. Therefore, the software process will consist of a continuous design and programming cycle. Recently, a number of 'agile' software development methodologies have strongly promoted the embrace of change, and the idea that applications will change after they have been created [1]. This acknowledgement is a

big step towards a new way of developing applications. Instead of trying to design the application to support all possible behaviors, even if not needed at present, you start with the simplest design possible and change it as needed.

For us, the way to build complex software is by doing it incrementally, with the simplest design possible.

The industry knows how to build object-oriented applications incrementally [1] [2], but there is no efficient way to achieve this when building the application's Data Layer. When a change in the object model involves a change in the database schema, it implies a number of manual and usually difficult steps to create the new schema, convert the data from the old schema to the new one, and modify the data access code to adapt to the new database schema. This usually discourages programmers, so they avoid making changes in the data model.

2. Building applications in multiple layers

The advantages of building applications in multiple layers are well-known in the industry. The basic idea of this architecture is to isolate the business logic and data access code from the user interface code. These middle tier components 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 can be centralized on server-side components that secure the data.
- Changes to the business rules only require you to update the server code, not each client.
- Middle-tier components can run on an application server, which provides runtime services such as connection pooling, to maximize the throughput and scalability of the applications.

As you can see there are many advantages to this architecture, but the difficulty lies in developing the middle tier components. This involves the tedious task of the object-relational mapping between your application objects and a relational database. These components are usually written in languages like C#, Java or Visual Basic and/or a database stored procedure language like T-SQL or PL/SQL.

3. The DeKlarit Approach

DeKlarit provides an innovative way to capture the knowledge needed to build an application. Instead of gathering the requirements and building an abstract design, the requirements are the only input DeKlarit needs.

Typically, end users are the ones who know how a business works, so DeKlarit's Business Components map to entities familiar to users. For example, if a user works with an Invoice, the information needed is:

- Invoice Number, Invoice Date, Customer ID, Customer Name

The invoice details will be:

- Item ID, Item Description, Price Per Item, Quantity, Line total

The invoice footer will be:

→ Invoice Total

If you use a traditional methodology, such as UML, the invoice design will be similar to this one:

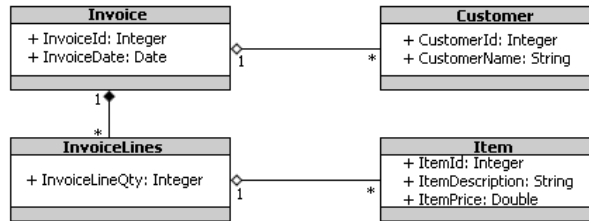


Figure 1

In DeKlarit, the same invoice will look as follows:

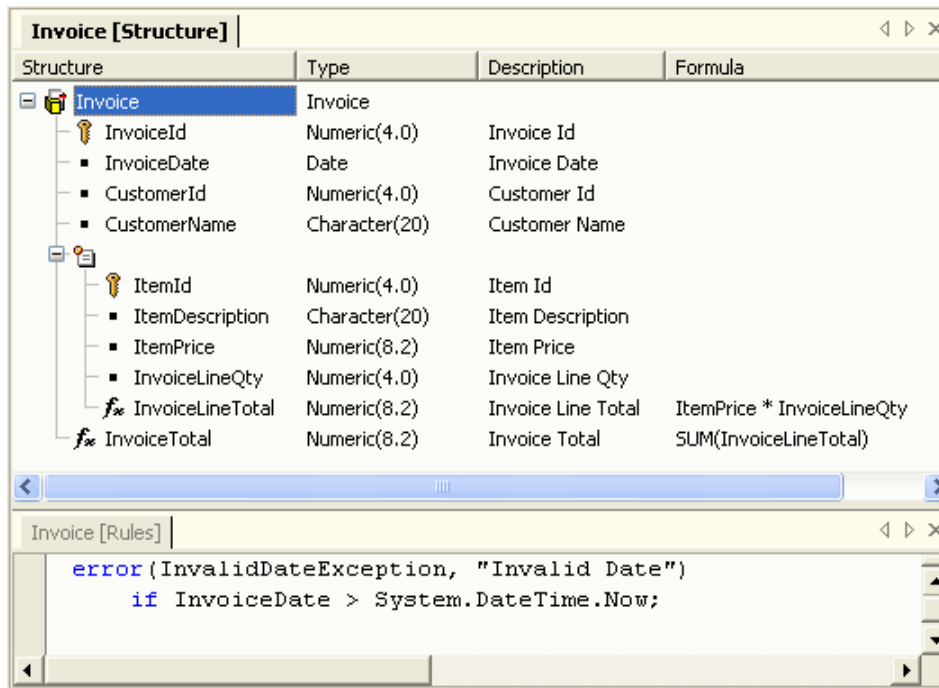



Figure 2

The attributes with a  identify the component (the primary keys).

The InvoiceLineTotal and InvoiceTotal are defined as calculated attributes. These attributes can also be set as 'redundant', and the redundancy will be automatically maintained. For example, if the InvoiceLineTotal is set as redundant, when the ItemPrice changes the InvoiceLineTotal must be updated. DeKlarit does this automatically.

The indented level implies a one-to-many relationship (one invoice has several invoice detail lines).

As you can see above, DeKlarit's invoice is much closer to reality than an UML diagram, and is something that the end user can relate to.

This invoice is defined as a **Business Component**, and is composed by a structure and a set of business rules. These rules are declared, not written in procedural code, so as to enable a higher level of definition of your business.

When coding the business rules in the middle tier components, the use of declarative rules has numerous advantages over procedural coding [3] [4]:

→ They are more closely related to user requirements, and that is why the coding is almost transparent.

- To change the application's behavior, all you have to do is change the business rules.
- Regarding the business logic, it lets you focus on 'what' not 'how'.
- You do not have to specify the order in which rules are triggered. The correct order is automatically determined by DeKlarit.

If you look at how the Invoice was defined, you will notice that the structure is **unnormalized**. This is why it looks closer to reality than the standard object model of the previous UML diagram. This point is very important to understand DeKlarit's philosophy, and the way it lets you design Business Components.

When building database-based applications you must create a normalized database schema. The existence of a normalization process implies that the real world entities look unnormalized. Through normalization we try to adapt reality to our model of reality, and this process involves **information loss**. In this example, the fact that the Invoice has a CustomerName attribute would be lost if the structure was normalized before making the information available to the data or object model. The lost information is very important, and its use is what makes you and DeKlarit much more productive. This does not mean that DeKlarit will define an unnormalized database schema, as a normalized database schema is the only way to achieve a consistent data model. What DeKlarit does is create a normalized data model based on the model's Business Components. Thus, it relieves you from the task of creating a normalized data model.

The relational model is the most efficient way to implement large, complex databases. One of the most common problems in software development is 'a badly designed data model', which results in data consistency problems. Usually, the data model is well designed at the beginning of the development phase, but it deteriorates during the application life cycle. DeKlarit helps you always have a good data model.

Going back to the example, after defining the invoice DeKlarit inferred these database tables:

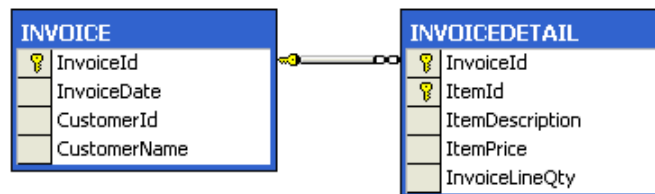


Figure 3

Now, if you add a Customer Business Component with the following structure:

Customer [Structure]			
Structure	Type	Description	Formula
Customer	Customer		
CustomerId	Numeric(4,0)	Customer Id	
CustomerName	Character(20)	Customer Name	

Figure 4

DeKlarit will change the schema to this:

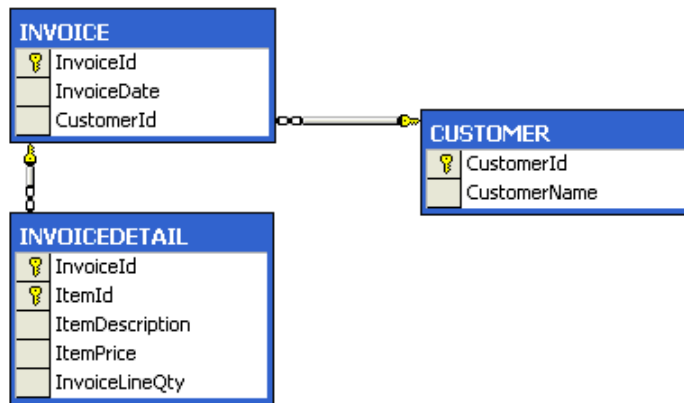


Figure 5

DeKlarit has defined a new database table, and has removed all the attributes that depend on the CustomerID from the Invoice table. In order to do this, DeKlarit assumed that the CustomerID attribute in the Invoice Business Component was the same as the Customer Business Component's, so it normalized the database schema using that information. This is one of DeKlarit's key design points: attributes with the same name refer to the same real-world elements.

At this point, DeKlarit has defined all these tables in its internal "working model", but not in any real relational database. With this information, DeKlarit will build an Impact Analysis report showing how it will create the database tables:

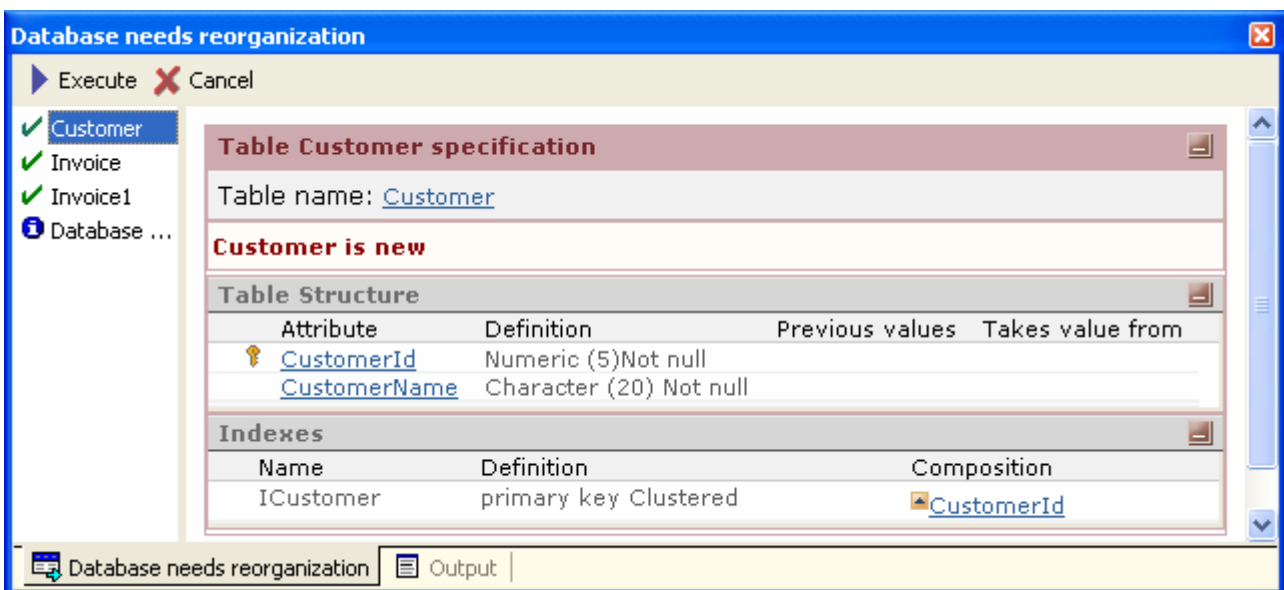


Figure 6

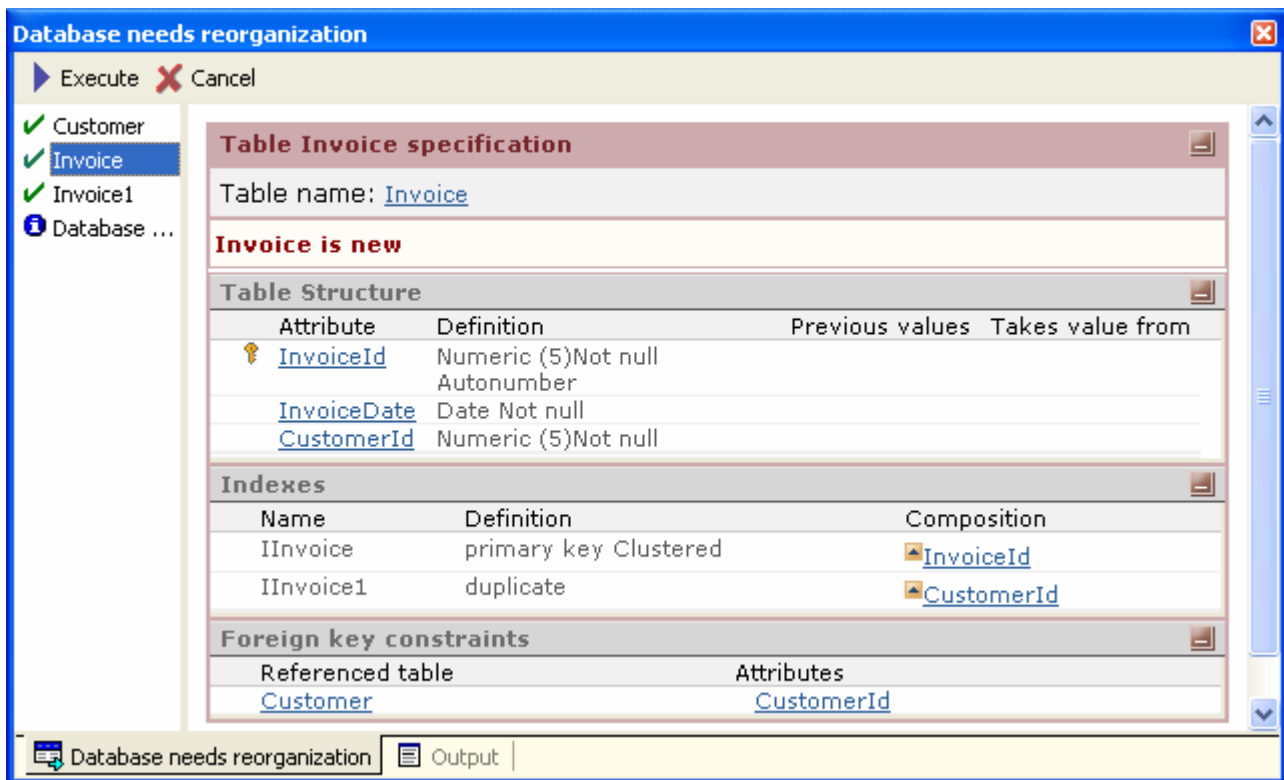


Figure 7

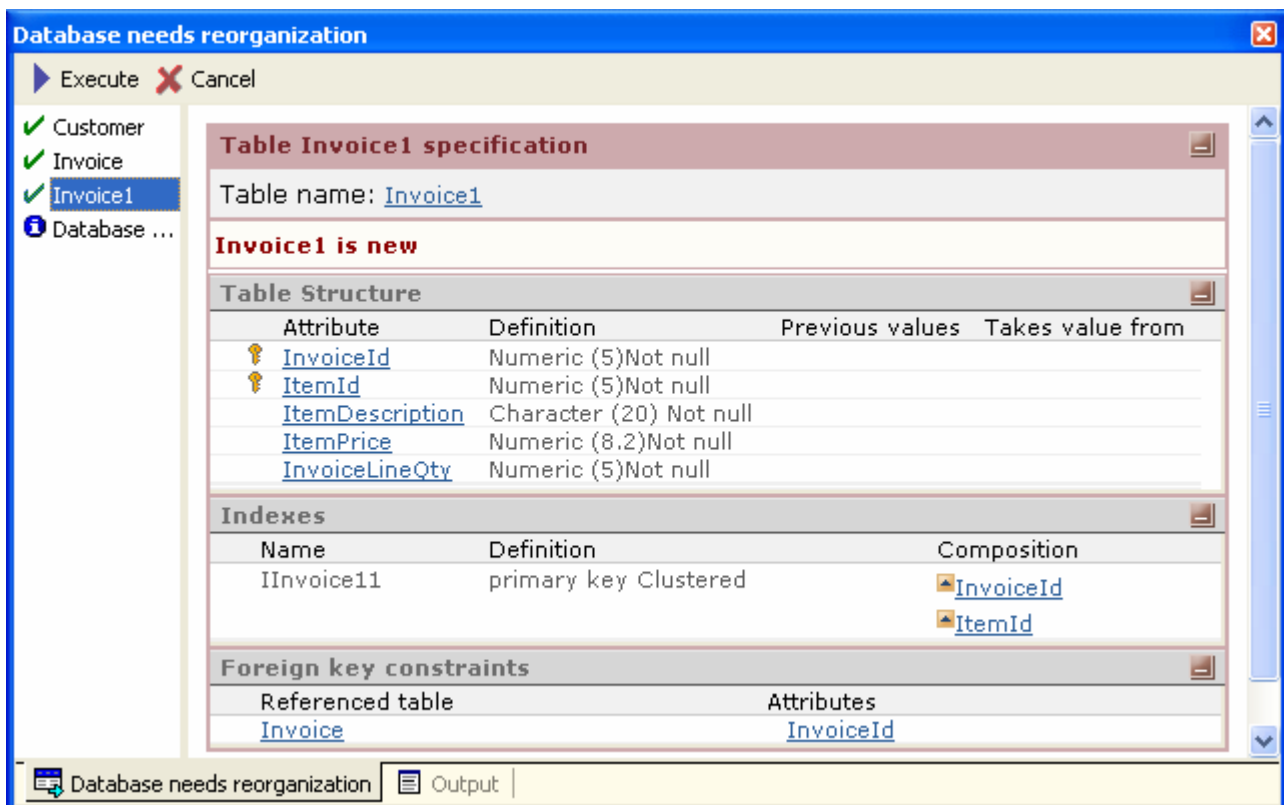


Figure 8

If you agree with the proposed structure, DeKlarit will automatically create the schema:

```

----- Compiling -----
Microsoft (R) Visual C# .NET Compiler version 7.00.9466
for Microsoft (R) .NET Framework version 1.0.3705
Copyright (C) Microsoft Corporation 2001. All rights reserved.

```

```

----- Executing -----
1. Locking tables to reorganize.
2. Dropping referential integrity constraints.
Dropping integrity constraints on table INVOICE.
Dropping integrity constraints on table INVOICEDetail.
Dropping integrity constraints on table CUSTOMER.
3. Create temporary and new tables.
Creating table INVOICE .
Creating table INVOICEDetail .
Creating table CUSTOMER .
4. Running reorganization programs.
6. Renaming tables.
7. Building indices and integrity constraints.
Creating index IINVOICE ...
Creating index IINVOICE2 ...
Creating index IINVOICE1 ...
Creating index ICUSTOMER ...
8. Dropping attributes and removed tables.

```

And then it will create a C# (or Visual Basic .NET, depending on the language you chose before executing the Build operation) project to generate the .NET components needed to work with the defined Business Components:

```

----- Analyzing -----
----- Specifying -----
Business Component : Invoice
Business Component : Customer

----- Generating -----
Business Component : Customer
Business Component : Invoice

----- Done -----

```

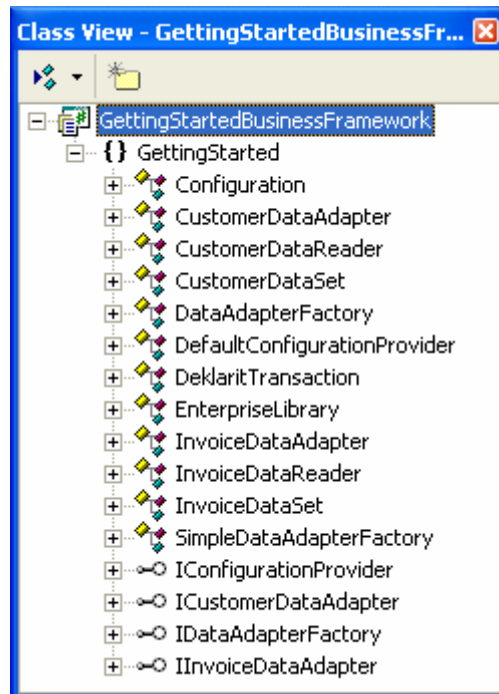


Figure 9

In this case, DeKlarit has generated C# source code for a DataSet, DataAdapter and DataReader for each Business Component. These classes are three of the main building blocks in ADO.NET. The DataSet has the Business Component data, defined in the same way as in the Business

Component structure (not as in the database schema). The DataAdapter knows how to retrieve, update and delete the data, while maintaining the referential integrity and applying the business rules. The business rule code is generated inside the DataAdapter. The DataReader provides a quick way to retrieve the Business Component's data.

Up to now DeKlarit has created a database schema with three tables, several database indices, two DataSets and two DataAdapters. All this was achieved by using the definition of those two Business Components.

A new Business Component called 'Item' will be added to the project in order to show how DeKlarit simplifies incremental development:

Structure	Type	Description	Formula
Item	Item		
ItemId	Numeric(4.0)	Item Id	
ItemDescription	Character(20)	Item Description	
ItemPrice	Numeric(8.2)	Item Price	

Figure 10

When this Business Component is created, DeKlarit normalizes the data model and builds a new schema for it:

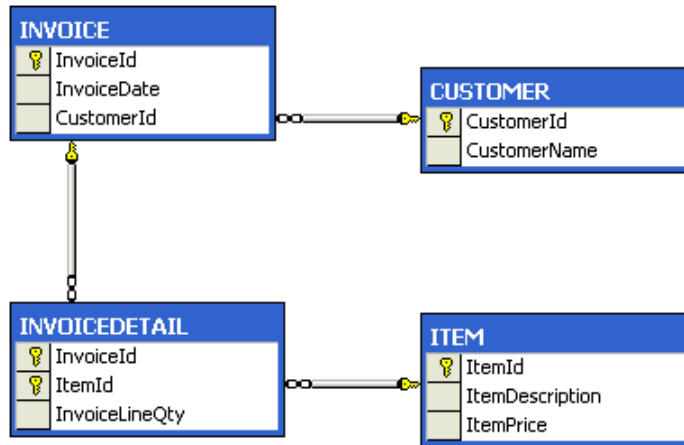


Figure 11

Now the ItemDescription and ItemPrice are no longer in the InvoiceDetail table but in the Item table. To change this manually you should create a new schema, write code for the programs to convert the data, as well as search, find, and fix all the programs that use the modified tables. DeKlarit does all this **automatically**.

How does DeKlarit handle database changes?

- It creates the new tables in the schema
- It converts the data from the old schema to the new one, so you do not lose your data
- It regenerates only the DataAdapters and DataSets that should change to map to the new schema.

The first step is to perform a new Impact Analysis:

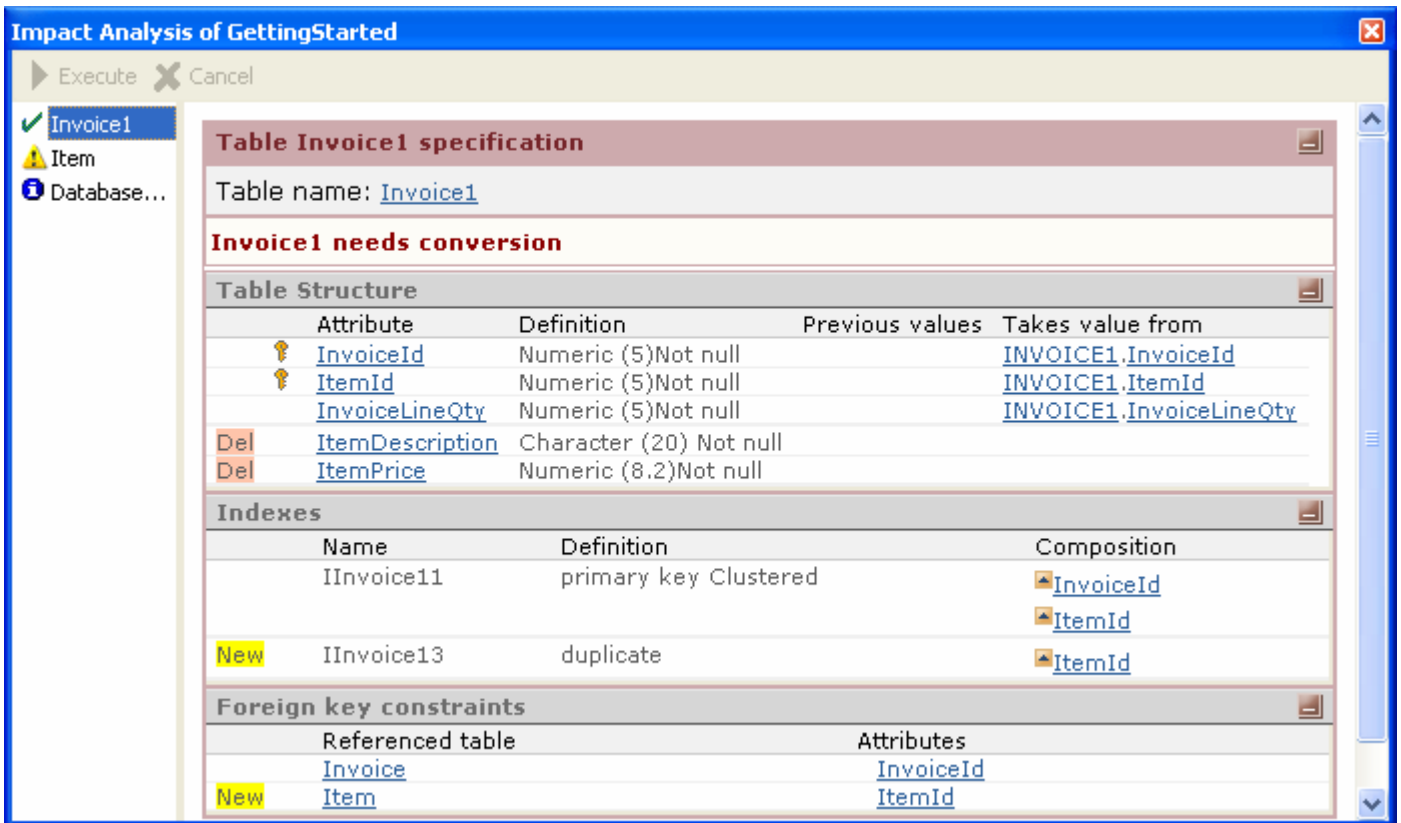


Figure 12

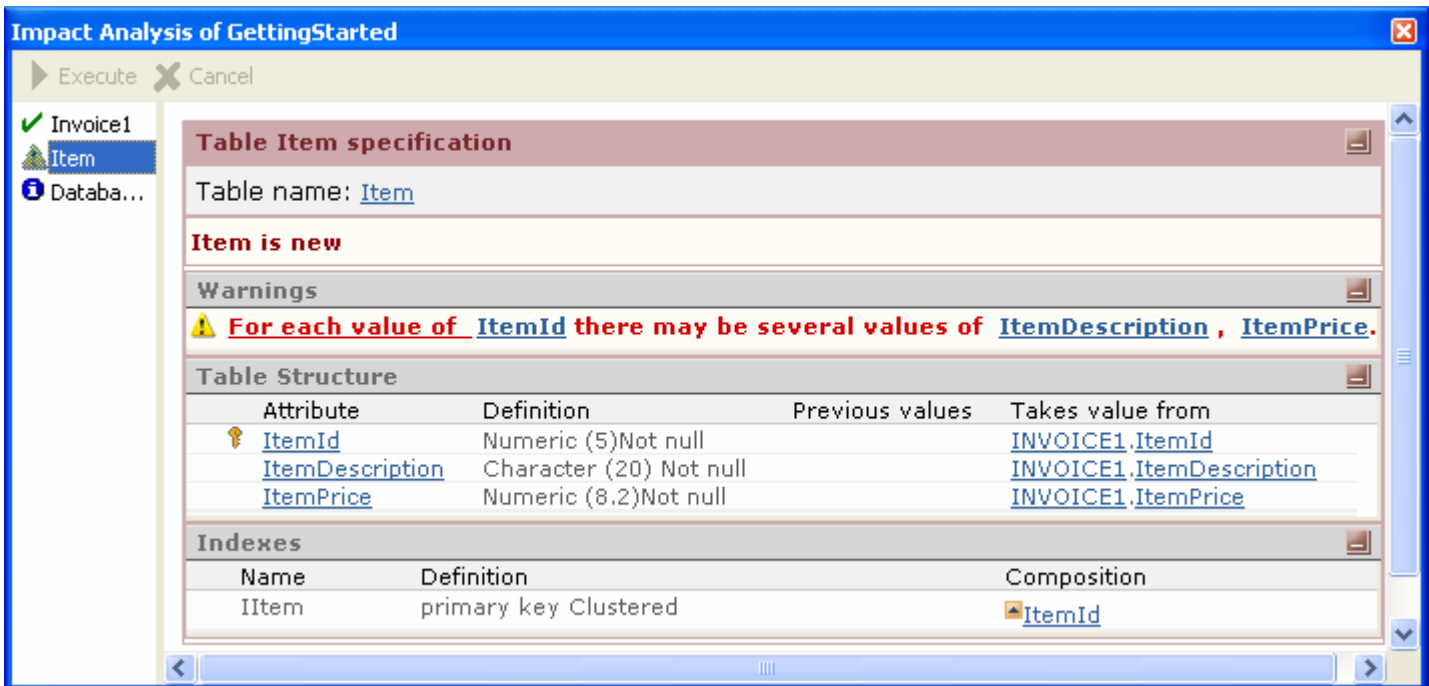


Figure 13

The Impact Analysis above shows that it will modify the InvoiceDetail table by removing two attributes and creating a new index. In addition, it will create a new Item table, taking the values from the InvoiceDetail table.

If you agree with the changes, the reorganization process is run:

```
----- Executing -----
1. Locking tables to reorganize.
```

```

2. Dropping referential integrity constraints.
Dropping integrity constraints on table ITEM.
Dropping integrity constraints on table INVOICEDETAIL.
3. Create temporary and new tables.
Creating table GXA0004 .
Creating index GXI4 (temp) ...
4. Running reorganization programs.
Running program C4.
Creating table ITEM .
Dropping index GXI4 (temp) ...
6. Renaming tables.
Renaming GXA0004 as ITEM
7. Building indices and integrity constraints.
Creating index IINVOICEDETAIL ...
8. Dropping attributes and removed tables.
----- Done -----

```

Then DeKlarit will regenerate the .NET components that have been impacted by the reorganization. In this example, they are the Invoice and Item Business Components.

```

----- Specifying -----
Business Component : Invoice
Business Component : Customer
Business Component : Item

----- Generating -----
Business Component : Invoice
Business Component : Item

----- Done -----

```

Even though the Invoice .NET component now accesses four tables instead of three, the **programmer classes that use that component do not need to be modified**. This is possible because the DataSet structure has not changed. For example, in a previous screen the following structure was defined:

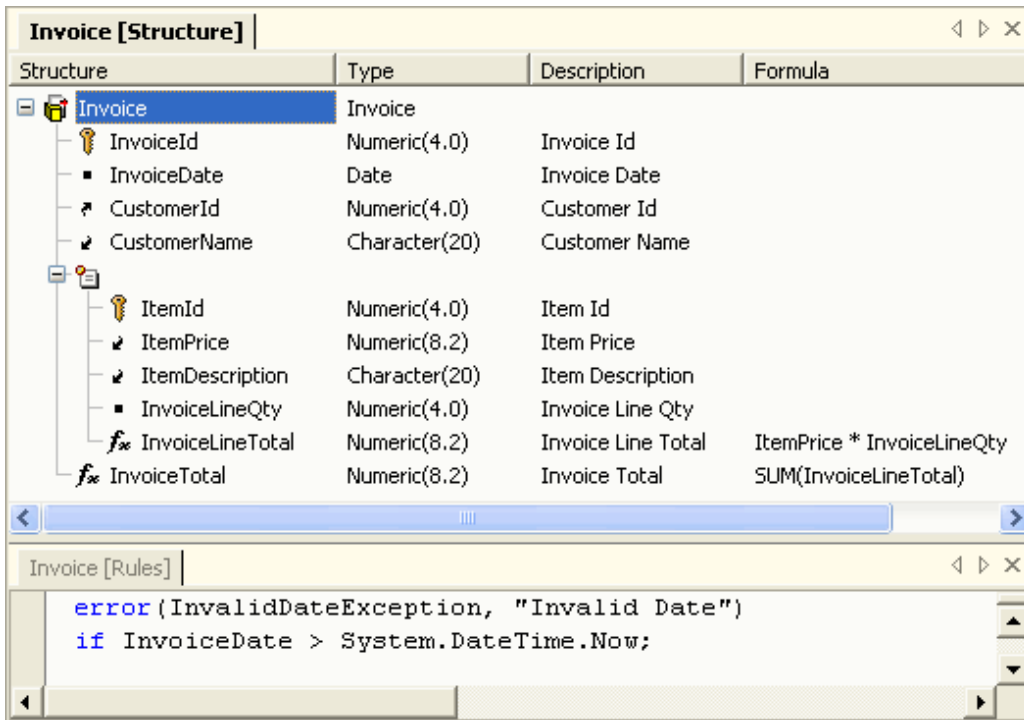


Figure 14

And it was mapped to:

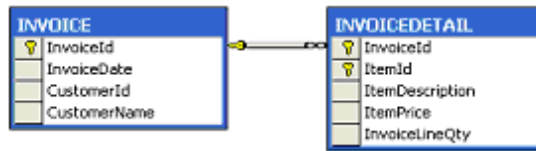


Figure 15

After adding the other Business Components (Customer and Items), the new schema looks like this:

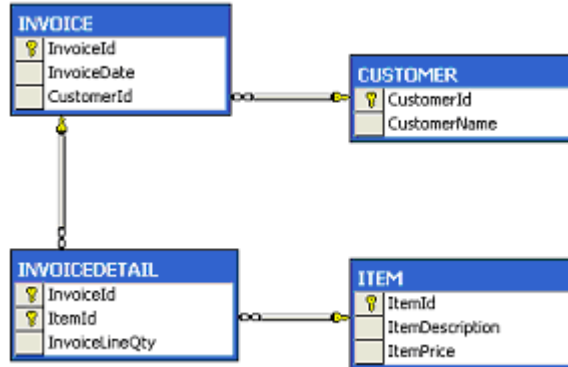


Figure 16

But the Invoice structure remains the same, and the programs that use it do not need to be changed! This is why defining Business Components as unnormalized structures gives great power to developers. In a traditional object model, the data layer classes need to be changed mainly because of the normalization process. This involves changing the definition of the data layer classes, and modifying the code that uses them. This process becomes unnecessary when the object model is unnormalized.

1.1. Retrieving Data

Business Components are designed to support transactional operations. Their objective is to insert, update and delete specific instances, but when it comes to retrieving data, a different view is needed.

If the data you need to retrieve is somehow related to the integrity relationships of the data model, the Business Components provide methods to retrieve the Business Components instances filtered by foreign keys. For example, the DataAdapter that corresponds to the Invoice Business Components has a `FillByCustomerId()` method that retrieves all the Invoices for a given Customer.

If the data to be retrieved is more complex, and involves attributes from many Business Components with specific orders and conditions, it can be retrieved by using Data Providers, which simplify the data retrieval process.

When an end user requests data, the 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. Using the DeKlarit **Data Providers**, the request can be defined in a very user-friendly way. This is simply done by selecting the attributes needed as shown in the example below.

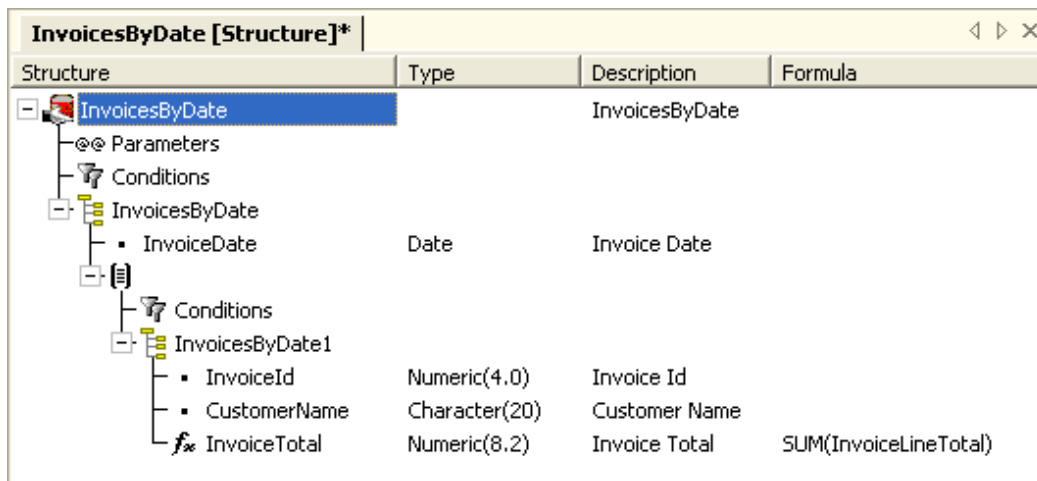


Figure 17

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, evaluate the appropriate formulas, and load the resulting data into the DataSet. As the Data Provider is specified without making reference to a specific database table, the programs that use the generated .NET components do not need to be changed when the underlying database schema changes.

4. Metadata-based Development

Together with the DataSets and DataAdapters, DeKlarit generates metadata that lets you build metadata-based applications. These applications can take advantage of the metadata at runtime, for example, by creating a data entry form for a specific Business Component and using the DataSet/DataAdapter to update the database, or to generate code based on that metadata. For example, a set of ASP.NET with Data Entry forms for the Business Components or with reports for the Data Providers. Samples of this kind of applications are included in the DeKlarit installation.

The ability to build Data Entry forms based on the metadata is also a consequence of the unnormalized way in which Business Components are defined. For example, if the object model knows that an Invoice has a Customer, but not which fields of that Customer, it is not possible to automatically build a Data Entry form that shows the Invoice information. You can take advantage of that information by telling the object model that the Invoice needs only the Customer Name.

5. Benefits of DeKlarit

Feature	Benefit
Simultaneous design of your components and database schema	The components are always synchronized with the database schema, and there is no need to write object-relational mapping code. Your components evolve together with the database schema.
Database normalization	DeKlarit helps you to always have a good data model, avoiding uncontrolled

	redundancies and data inconsistency. You do not need expertise in Relational Database Modeling to create a good design.
Business Rules	You can declare your business logic in the Business Components, which makes it very easy to specify and maintain their behavior.
Database refactoring & database schema independence	You do not need to worry about the consequences of changing your database schema. You can modify it as needed because you are not "locked-in-time" with your initial data model. Your applications are more agile to adapt to new requirements.
Code generation	You save a lot of development time using reliable components that do all the data validation, persistence and retrieval.

6. References

1. Kent Beck: *Extreme Programming Explained: Embrace Change* (1999, Addison-Wesley Pub Co; ISBN: 0201616416)
2. Martin Fowler, et al.: *Refactoring: Improving the Design of Existing Code*, (1999, Addison-Wesley Pub Co; ISBN: 0201485672)
3. C. J. Date, *What Not How: The Business Rules Approach to Application Development* (2000, Addison-Wesley Pub Co; ISBN: 0201708507)
4. Ross, Ronald: *Business Rules Concepts – The new mechanics of Business Information Systems* (1998, Business Rule Solutions, Inc, ISBN: 0-941049-04-3)
5. Hugh Darwen, C J. Date: *Foundation for Object / Relational Databases: The Third Manifesto* (1998, Addison-Wesley Pub Co; ISBN: 0201309785)