

# Object-Oriented Development in a Relational World: Applying DeKlarit

Chris Sells and Chris Tavares

The relational database (RDBMS) has earned its place as a key part of our modern technological society. The RDBMS is an essential part of almost every business's back office, from the cash registers at K-Mart to the largest accounting systems at the IRS. It could be argued that without the RDBMS, the web never would have become more than a display of static pages. Would amazon.com be able to go from my clicking that ever-so-seductive "1-click order" button to a box of comic books on my doorstep without a robust, flexible and reliable way to store all the millions of pieces of data that go into running any organization that large?

However, like all technology, the power and flexibility of the RDBMS comes with a price: the relational model itself. The relational model requires all data be represented as simple data types collected into tables, delineated by fields into columns and by identity into rows. Our new-fangled object-oriented view of the world is torn asunder when we're forced to replace types with tables, pointers with foreign keys and collections with rowsets.

And, like object-oriented design, there's a right way and a wrong way to design databases. This process, called "normalization," while great for the consistency and integrity of the data itself, pulls us even further from any semblance of object-oriented design we might have hoped to maintain. As a result, the industry has given birth to specialized engineers to design and maintain databases who are often far-removed from application development, both in expertise and in sympathies. Like many such couples, the relationship between application engineer and database engineer is as fraught with strife as that of husband and wife, development and marketing, or even, "tastes great" and "less filling." In short, the mapping of the needs of the application to those of the database is often an arduous and time-consuming part of any project that needs both.

As an example, one of the authors of this article (Chris) is the webmaster for a local game convention. For those who aren't familiar with them, the big difference between a developer's convention and a gamers' convention (other than the subject matter, of course) is that a game convention has a large number of small events, typically with no more than ten people at each. As a result, scheduling events at a game con can be quite challenging. Another difference is the staffing and budget at a game con, or rather, the lack thereof. As a small nonprofit group with a minimal budget, the primary means of communication with game con attendees is through a web site.

When Chris (not Chris) took over the game con web site, the biggest problem was that the site was strictly static. This prevented attendees from signing up for their favorite games and prevented staffers from being able to do queries to filter events. But even worse from Chris's perspective, updating the event lists were an exercise in frustration. The event coordinator would schedule a new event by sending a description to the publications person, who would format it for the program book. Chris would then have to manually pull stuff out of the program book Word document, reformat everything, and finally get it on the web. Or, to put it another way, the entire process was a gigantic pain in the behind.

So, this year Chris decided to fix these problems using ASP.NET and ADO.NET. A dynamic database-driven site would allow him to store all the event information in one place, simplifying the production of the web site and the program book. It would save the time of having to update the site every time a new event is scheduled, it would give staffers a way to run queries and it would open up the chance to register online for games. The only real problem was time. While Chris was very familiar with ASP.NET and ADO.NET, Chris (the one maintaining the site) was not, so the learning curve was a bit steep, especially for a wholly volunteer activity. The goal, of course, was to get the best site in the least amount of time, so that Chris could get on with actually playing the games rather than scheduling them.

# Designing the Event Management System

For the convention web site, Chris identified the following functions that need to be performed:

- ?? Build the convention event listings from a database so attendees can see what games are being offered.
- ?? Allow the convention staff to generate reports from the database.
- ?? Allow registered attendees to sign up for convention events online.

The first of these (the convention event listing) was highest on the list, since it represented most of the wasted time on the old site. Looking at last year's program book, a convention event can be expressed as the following C# type:

```
public class ConventionEvent {
    public int      EventID;
    public string   EventName;
    public DateTime StartTime;
    public string   EventLocation;

    public enum Experience { None, Familiar, Expert };
    public Experience ExperienceRequired;

    public string RefereeName;
    public int    RefereeID;
    public string Description;
};
```

This translates fairly easily into a table in a database (assuming we're willing to be a bit liberal about the type mappings):

```
CREATE TABLE ConventionEvent (
    EventID          int          primary key,
    EventName        varchar(64),
    StartTime        datetime,
    EventLocation    char(8),
    ExperienceRequired smallint,
    RefereeName      char(80),
    RefereeID        int,
    Description      varchar(1024)
)
```

This table alone is enough to produce the convention's web site and the program book, as well as allow convention staff to do their queries (assuming they can speak SQL), but it still doesn't allow for attendees to sign up. And what's an attendee anyway? Based on past experience, here's what we need to know about game con attendees:

```
public class Attendee {
    public int    BadgeNumber;
    public string AttendeeName;
    public bool   IsPaid;
};
```

This translates into a similarly pedestrian database table:

```
CREATE TABLE Attendee (
    BadgeNumber int          primary key,
    AttendeeName char(32),
    IsPaid      bit
)
```

All we need now is some way to hook the two tables together so that we can tell who's attending what event. One possibility would be to change ConventionEvent to this:

```
public class ConventionEvent {
    public int      EventID;
    public string   EventName;
    public DateTime StartTime;
    public string   EventLocation;
};
```

```

public enum Experience { None, Familiar, Expert };
public Experience ExperienceRequired;

public int RefereeID;
public string RefereeName;
public string Description;

public int NumSeats;
Attendee[] attendees;
};

```

Here's where the application engineer and the database engineer butt heads. Developers looking at this definition are nodding their heads at this definition, but database folks are cringing. Databases don't handle things this way. Instead, you use a separate table that has the event ID and the attendee ID as fields, and then you use a join at run time to figure out who's signed up for what. However, from an application standpoint, I don't care what the database looks like, and having to worry about it on this kind of project is time better spent on other parts of the site.

This kind of project cries out for a tool that will take my simple data structures and figure out how to read and write them to and from the database, creating relationship tables as appropriate. And while I'm wishing, as I refactor my structures, such a tool would also be smart enough to rearrange the tables (and the data!) appropriately.

Luckily for Chris, who's got to get this site up and running, such a tool exists. It's called DeKlarit (available from <http://www.DeKlarit.com>).

## Defining Data in DeKlarit

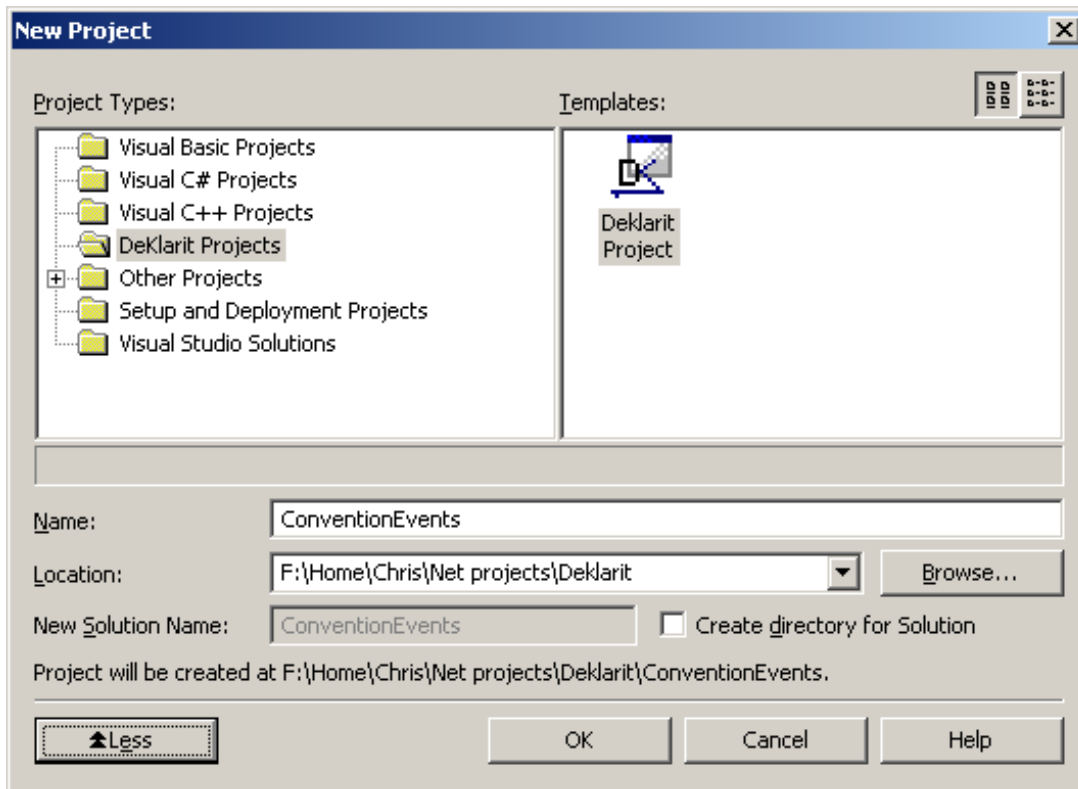
ARTech, the makers of DeKlarit, list the following features on their web site:

- ?? DeKlarit integrates into Visual Studio .NET
- ?? No need to write code for the Business or Data Layer - just describe your Business Components!
- ?? When something changes, the database schema and .NET components are recreated automatically and the data is spread to the new schema
- ?? DeKlarit automatically generates the Database and Business Components

Instead of being forced to work from the normalized data structures that an efficient database requires, DeKlarit allows you to define whatever structures are convenient for you. It takes care of the mapping between your data structures and the normalized ones needed by the database.

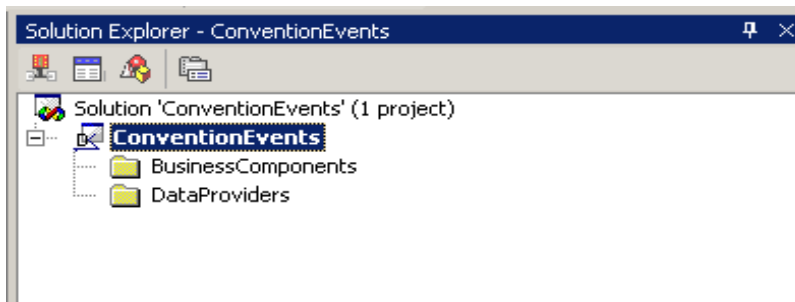
While this is sufficiently "markety" to make one suspicious, it sounds like DeKlarit is ideal for our needs, so let's give it a try.

After setup, the first step in using DeKlarit is to create the project in Visual Studio .NET (VS.NET), as shown in Figure 1.



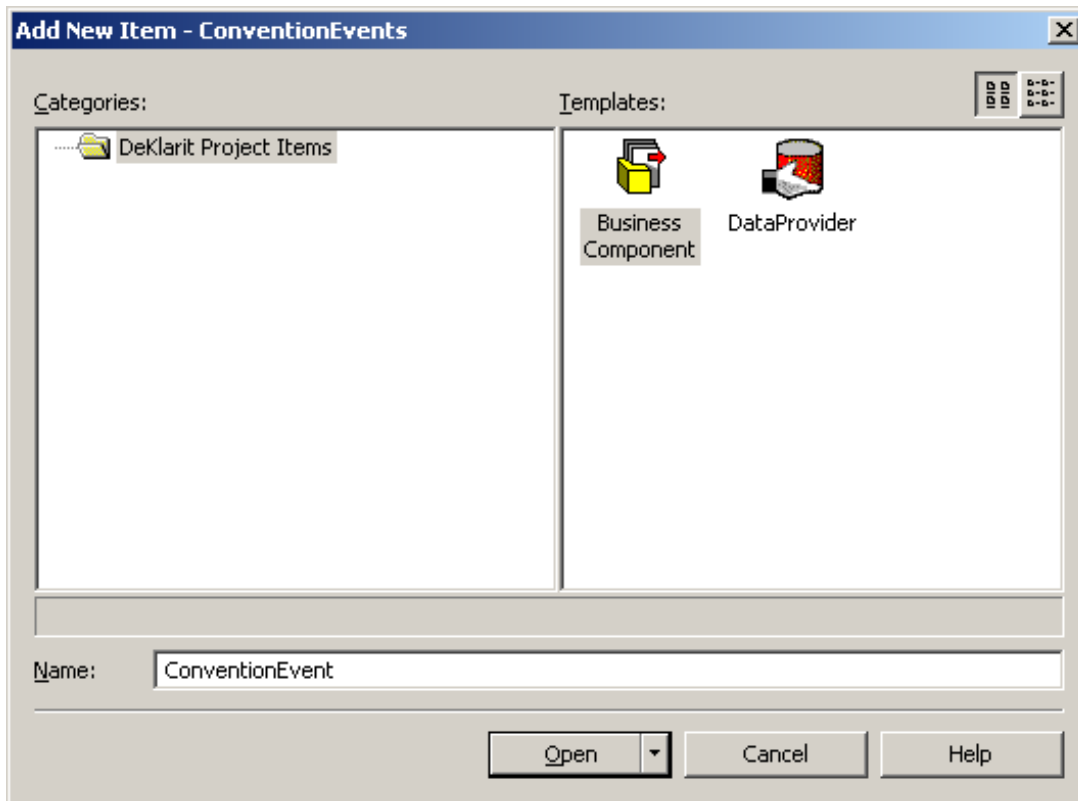
**Figure 1 - Creating a DeKlarit Project**

This will create a new project, laid out as in Figure 2.



**Figure 2 - A DeKlarit solution**

Out of the box, this doesn't really do anything since we haven't included our data yet. So, let's go ahead and define our event structure. Right click on the solution, and choose "Add Item" like you would in any VS.NET project. This brings up the usual Add New Items box (as shown in Figure 3). DeKlarit has two kinds of items you add to its projects: Business Components and DataProviders. Business Components correspond to the structures we defined above in the C# code. Data Providers are "views" on existing Business Components and we'll look at these later. We start with our convention event Business Component.



**Figure 3 - Adding our convention event Business Component**

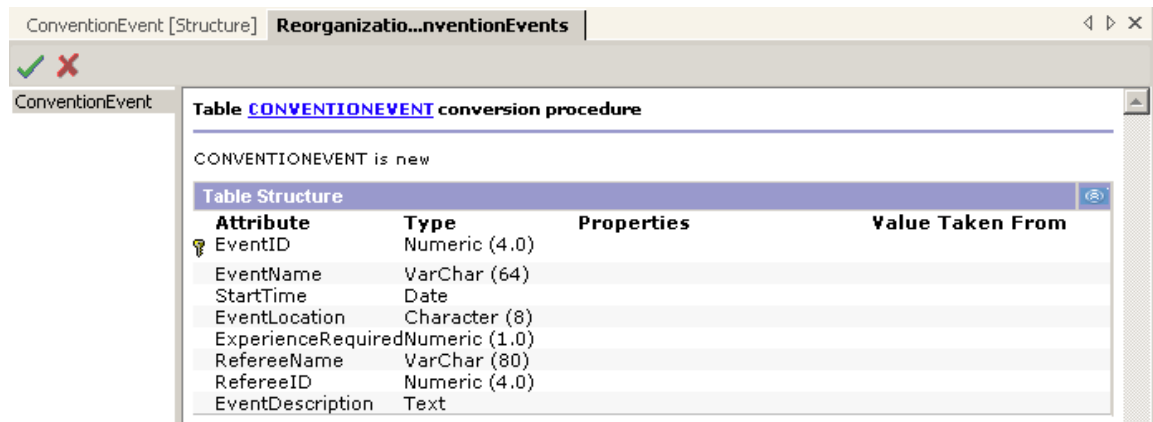
Now we get to the meat of the project – defining the fields of the convention event. Starting with the simpler ConventionEvents structure above, entering the fields yields the result in Figure 4.

ConventionEvent [Structure]			
Structure	Type	Description	Formula
ConventionEvent		ConventionEvent	
EventID	Numeric(4.0)	Event ID	
EventName	VarChar(64)	Event Name	
StartTime	Date	Start Time	
EventLocation	Character(8)	Event Location	
ExperienceRequired	Numeric(1)	Experience Required	
RefereeName	VarChar(80)	Referee Name	
RefereeID	Numeric(4.0)	Referee ID	
EventDescription	Text	Event Description	

**Figure 4 – Defining the Initial Event Structure**

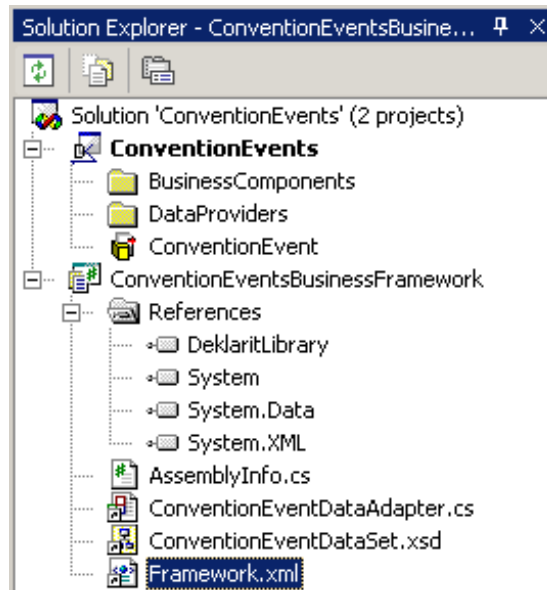
This is essentially the same thing as we had above in our initial design. Each field in our structure was entered as an attribute of ConventionEvent. The Type column lists the data type, and the dropdown box makes it easy to choose types and sizes. The DeKlarit help file has more details on how its data types map to the database.

So, let's build this and see what DeKlarit gives us. First, we set the `ConnectionString` property as we would with any database app, and compile. The first thing DeKlarit does it generate an "Impact Report", telling us what will happen to our database. In this case, I started with an empty database, so DeKlarit went ahead and created the `ConventionEvent` table in the database (see Figure 5).



**Figure 5 - Convention Event component report**

DeKlarit created the table in SQL Server, setting up the columns as we defined in our structure. What else did it do? Take a look at Figure 6, which is the solution after compiling the DeKlarit project.



**Figure 6 - Solution after compiling ConventionEvents**

Notice the new `ConventionEventsBusinessFramework` project. This contains C# code that implements an ADO.NET data adapter class and a dataset class, encapsulating everything about the database into type-safe classes that can be used anywhere you'd do ADO.NET coding. For example, here's a simple console program that generates a text-only rendition of our event listing:

```
using System;
using System.Data;
using ConventionEvents;

class App {
    static void Main(string[] args) {
        // The DeKlarit generated data adapter class
        ConventionEventDataAdapter da = new ConventionEventDataAdapter();
        // And the corresponding dataset class
```

```

ConventionEventDataSet ds = new ConventionEventDataSet();

// Pull the data out
da.Fill( ds );

ConventionEventDataSet.ConventionEventDataTable dt = ds.ConventionEvent;
foreach( ConventionEventDataSet.ConventionEventRow row in dt.Rows ) {
    Console.WriteLine( "-----" );
    Console.WriteLine("Event {0}: {1} {2} Table {3}",
        row.EventID, row.EventName, row.StartTime, row.EventLocation );
    Console.WriteLine( "    {0} Experience required: {1}",
        row.RefereeName, ExperienceToString( row.ExperienceRequired ) );
    Console.WriteLine( row.EventDescription );
}

static string ExperienceToString( int exp ) {
    switch( exp ) {
        case 0: return "None";
        case 1: return "Familiarity with rules";
        case 2: return "Expert players only";
        default: throw new ApplicationException("Oops!");
    }
}
}

```

As simple as this will make displaying our events to convention attendees, there's still the matter of providing an administration interface to convention staffers. It doesn't have to be pretty, but it does require a fairly fully featured UI if we're going to break them of the habit of sending around Word documents. As if the DeKlarit folks knew exactly what Chris was up against, they provide a feature that will automatically build a web project for you with simple forms that let you add, edit, or delete records in your defined structures. All you need to do is compile the Business Framework project, then right-click on it in the Solution Explorer and pick "Create Web Project". Deklarit will connect to your web server, and put all the pieces of a usable ASP.NET project there. Figure 7 shows what the generated web page looks like after adding a couple of events.

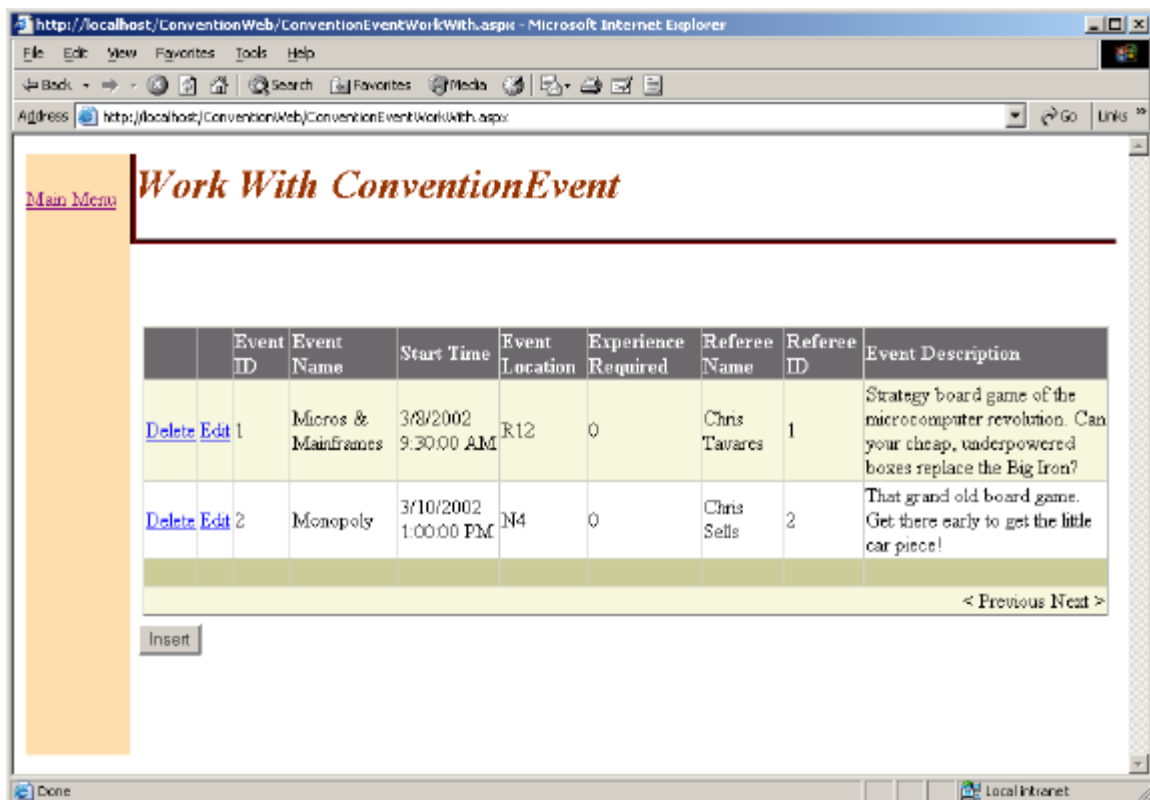


Figure 7 - The generated editing page

This page isn't the prettiest, but it works. The point of DeKlarit isn't really to provide application generation functionality, after all, but it's very convenient for quick tests to have this function. And, if you want to, the source code for the web project generator is included with DeKlarit for you to tweak the output.

## Relating Tables

One of the things that every game convention needs to do is keep referees happy. Without them, you have no events and no con. So, a reasonable report would be "Who's running what event?" And more importantly, "How many events is each referee running?" This way the staff can track who their best referees are and reward them (with fame & thanks, if not cash).

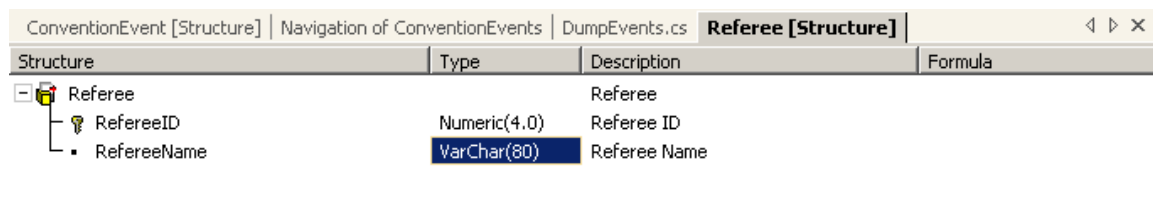
Let's take a look at the data again. The RefereeID and RefereeName fields stand out. From an OO standpoint, these two fields represent a separate entity from an event. They should be pulled out into a separate structure to get a cleaner design. This also puts the information into a separate table in the database, making it easier to do the reports.

If we were doing this project as a normal database project, what would we need to do to move Referees into their own table? We'd need to:

- ?? Define the new table.
- ?? Change the existing references to Referee fields in other tables to just reference the primary key of the new table.
- ?? Migrate the data out of our old table structure into the new one.
- ?? Create SQL joins to put everything back together.

Looking at that list, the first one seems fairly easy, but the rest of them involve quite a bit of work, particularly the data migration.

When using DeKlarit, you code against Business Components, not database tables. Let's see how it handles this situation. We add a new Business Component, Referee, with RefereeID and RefereeName fields (see Figure 8):



The screenshot shows the DeKlarit IDE with the 'Referee [Structure]' tab selected. The interface displays a tree view on the left and a table on the right. The tree view shows a 'Referee' component with two sub-fields: 'RefereeID' and 'RefereeName'. The table on the right lists the fields with their types and descriptions.

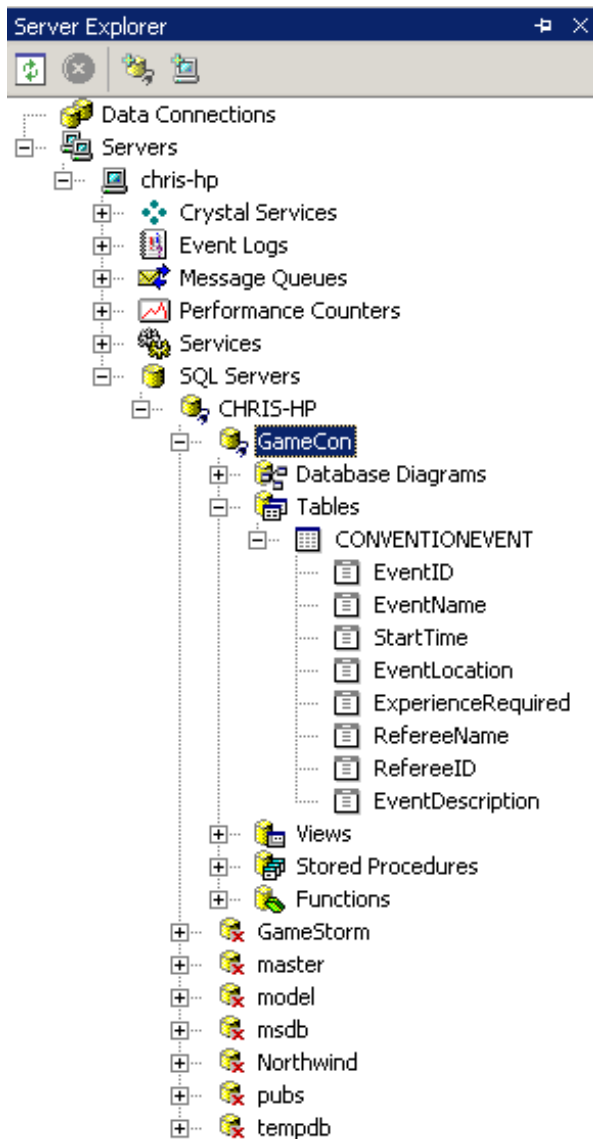
Structure	Type	Description	Formula
Referee		Referee	
RefereeID	Numeric(4,0)	Referee ID	
RefereeName	VarChar(80)	Referee Name	

**Figure 8 - Referee Business Component**

And, well, that's it, right? From our OO point of view, the referee information should still be accessible from the ConventionEvent. So what happens to the database?

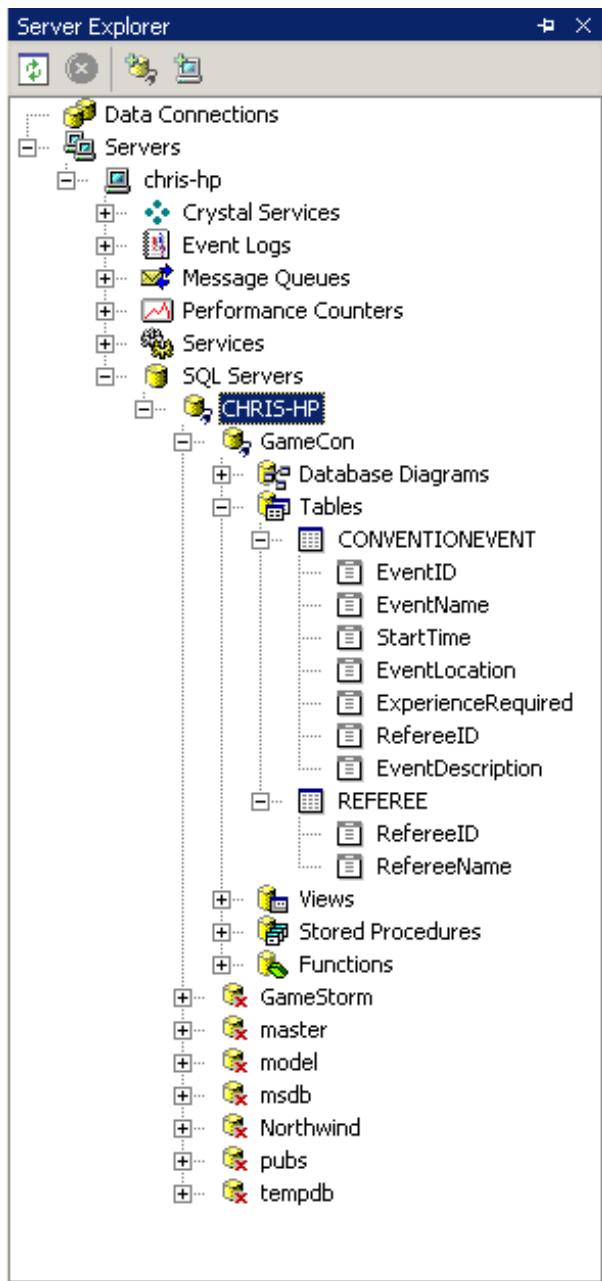
Before we rebuild the DeKlarit project, let's take a look at the current state of the database table. The server explorer currently shows this:





**Figure 9 - Tables before Adding Referee**

When we compile, DeKlarit shows us a screen indicating that the database needs to be reorganized and what needs to be done. Accepting the reorganization, let's take a look at Figure 10, the state of the database after the build:



**Figure 10 – Tables After Adding Referee**

Notice that the Referee table got added, just as the ConventionEvent table was added the first time around. But notice that ConventionEvent table has been updated; specifically, the RefereeName field is gone. And, if you look at the contents of the new Referee table, you'll notice that everything that had been in the old RefereeName and RefereeID fields has been automatically migrated to the new table structure! No more writing one-shot data reorganization code!

But that's not all. Recompile the ConventionEventsBusinessFramework project to update the data adapters and rerun our dumper program without recompiling it. It still works. The schema has changed significantly, and yet our access code doesn't change at all.

How does this work? The important thing is that you write your code against the Business Components, not the database schema. As a result, as you tweak things to make the database more efficient or add data structures, your old code continues to work as long as you don't delete a field your code was using.

How does DeKlarit get the RefereeName into the ConventionEvent? Because RefereeID is the primary key for a structure elsewhere, DeKlarit's generated code for the ConventionEvent structure will do the join to pull the RefereeName out of the Referee table. This provides the programming convenience we want, while keeping the database properly normalized.

The obvious question here is "How does DeKlarit know what the primary keys are?" It follows a simple rule – all attributes within a project that refer to the same thing must have the same name. If you use RefereeID anywhere in any structure, that attribute will always refer back to the primary key of the Referee structure. So how does DeKlarit know which table to join to? The secret is in a little icon in the Structure definition editor. Notice the little key icon. This indicates that this field is the primary key for this structure. If you reference an attribute in a structure that is a primary key in another structure, it will automatically recognize and create the join. If there are additional attribute references in that same structure, it will join on that primary key.

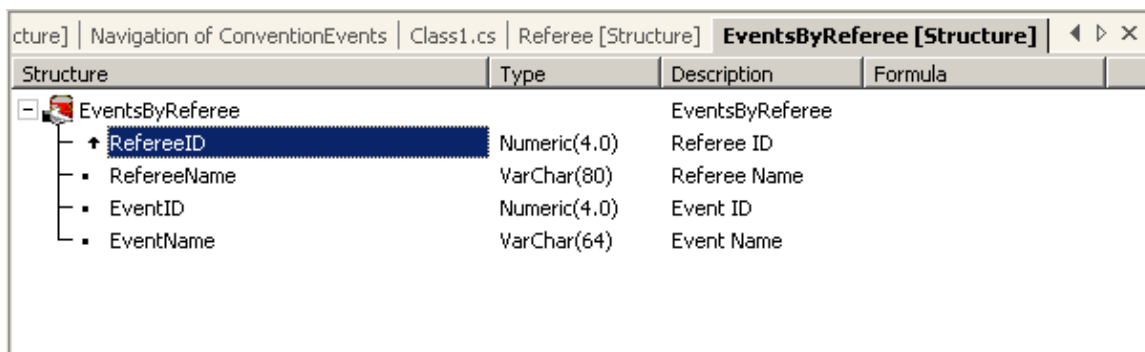
Also, in addition to keeping our data properly normalized, DeKlarit is even smart enough to know which table things are defined in. Notice that both the ConventionEvent and the Referee structure have a RefereeName field in them, yet magically, DeKlarit managed to figure out which table actually contained the RefereeName data. In this case, DeKlarit noticed that while ConventionEvent refers to Referee, the inverse is not the case. So, while all RefereeName fields in all structures must, by DeKlarit edict, refer to the same data, DeKlarit has figured out that the data must reside in the Referee table; otherwise, both Referee and ConventionEvent couldn't access the data. If, on the other hand, ConventionEvent didn't have a reference to the Referee table (via the RefereeID foreign key), DeKlarit would give us an error, because there would be no way to keep the data appropriately normalized.

In other words, not only is DeKlarit figuring out where to put the data based on the structures we define, thereby keeping the applications guys happy, it's also checking that these structures only allow normalized data, thereby keeping the database guys happy. The marketing hype from the web site -- "No need to write code for the Business or Data Layer - just describe your Business Components!" -- is starting to sound like hype less and less...

## A Change of Perspective

Still, as cool as DeKlarit is, we still haven't actually implemented those reports. One of the most useful features of an RDBS is the flexible reporting available. With a simple (or not so simple) SELECT statement, it's possible to fold and spindle (but not mutilate) your data into almost any form imaginable. But you need to be able to write that SQL.

DeKlarit also lets you define views of your data. You do this by adding a Data Provider component to your DeKlarit project. Then you define your view using an editor very similar to the business component editor, as shown in Figure 11.



cture]   Navigation of ConventionEvents   Class1.cs   Referee [Structure]   <b>EventsByReferee [Structure]</b>   ◀ ▶ ✕			
Structure	Type	Description	Formula
EventsByReferee		EventsByReferee	
↑ RefereeID	Numeric(4,0)	Referee ID	
▪ RefereeName	VarChar(80)	Referee Name	
▪ EventID	Numeric(4,0)	Event ID	
▪ EventName	VarChar(64)	Event Name	

**Figure 11 - EventsByReferee view**

Note the arrow icon next to the RefereeID field. This indicates that the view is sorted in ascending order on RefereeID; you can choose to sort by ascending or descending order, or to not sort at all. DeKlarit

generates a new data adapter and typed dataset class when you build the DeKlarit project. Here's a C# snippet that uses the newly defined Data Provider and spits out a report of Referees and events:

```
void DumpByRef() {
    EventsByRefereeDataAdapter da = new EventsByRefereeDataAdapter();
    EventsByRefereeDataSet ds = new EventsByRefereeDataSet();
    da.Fill( ds );

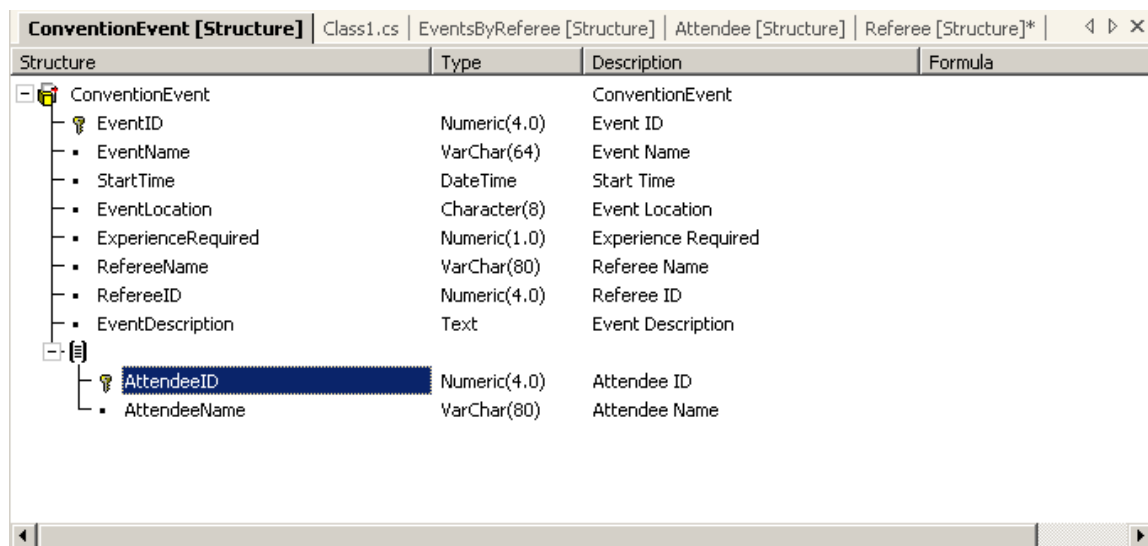
    EventsByRefereeDataSet.ConventionEventDataTable dt = ds.ConventionEvent;
    foreach( EventsByRefereeDataSet.ConventionEventRow row in dt.Rows ) {
        Console.WriteLine( "Referee: {0}, Event: {1}", row.RefereeName, row.EventName );
    }
}
```

As you can see, using the Data Provider is almost identical to using any other DeKlarit structure. You can also add formulas and sublevels to Data Providers just like you can to Business Components (this is discussed below). DeKlarit takes care of building the SQL SELECT statements for you.

## One to Many

So far, we've been able to use DeKlarit to effectively model our data, provide for populating the web site and for the staffers to run their reports, but we haven't done one very important thing: let attendees actually register for convention events. When most folks show up to the single game of Parcheesi and only one hairy guy shows for Strip Poker, things are going to get ugly (sic).

What we did with the referee information is implement a one-to-one relationship, i.e. each ConventionEvent has exactly one Referee. Now, we need a one-to-many relationship to map ConventionEvents to attendees. DeKlarit allows us to do this fairly easy with multiple level structures. We simply add an array of attendees who have signed up for the event to the ConventionEvent, as shown in Figure 12.



**Figure 12 - Adding Seats**

And, since it's fairly obvious to me that I'm going to need to treat attendees as independent entities for other purposes, I'll go ahead and define an Attendee Business Component as well, with AttendeeID and AttendeeName fields. Build this, and we see that DeKlarit has again reorganized our database, and automatically created the auxiliary table needed to properly handle the one-to-many relationship. And our original database dumper still works...

## Subtypes

Now that we have a separate Attendee structure, it's a good idea to reconcile an issue we have with our data model. Right now, we've got them separate, but really a referee is an attendee (albeit one that has to work part of the time), so there really shouldn't be a Referee structure at all. What makes an Attendee a referee is the RefereeID in the ConventionEvent structure. However, since DeKlarit figures out foreign keys by name alone, how do we get RefereeID to refer to our Attendee structure? Certainly changing RefereeID to AttendeeID doesn't work, since we've already got an AttendeeID in there to list event attendees. Even if we could change RefereeID to AttendeeID, we wouldn't want to, as that would take away the semantic meaning we derive from the relationship. So, how do we get DeKlarit to match fields with different names? Subtypes.

A subtype is a group of attributes that are aliases for other attributes in the system. Let's start by deleting the Referee component, since it's no longer needed. Then, we define the Referee Subtype:



Figure 13 - Adding a SubType

As a matter of good design, if you're using a subtype, don't use the parent type's fields in the same structure, i.e. since we're using RefereeID in ConventionEvent, we shouldn't also be using AttendeeID (Attendee is a super-type of Referee). In this case, it's helpful to define another subtype group, e.g. Player, with PlayerID and PlayerName as attributes. This way, it's blatantly obvious what each role is in the structure. The new ConventionEvent structure is shown in Figure 14.

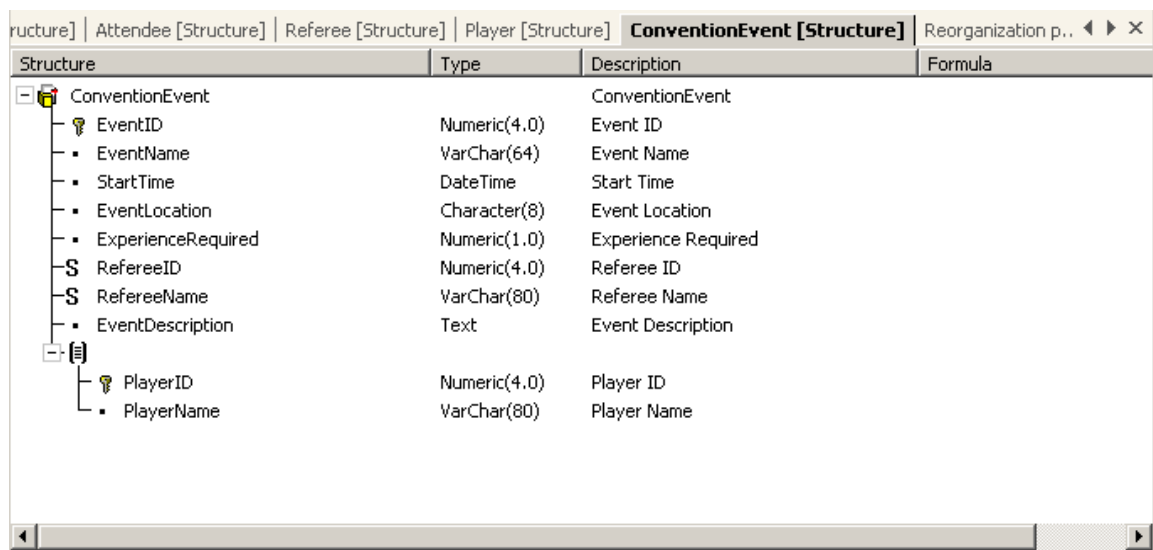


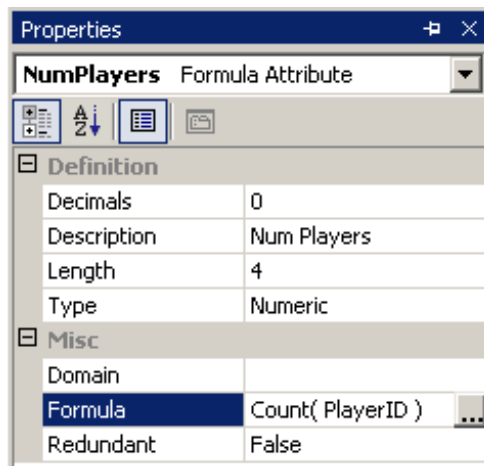
Figure 14 - Using subtypes

## “500 people signed up for Monopoly?”

Since we now have a way to sign people up, we need to have a way to prevent people from signing up, at least, we need to prevent too many people from signing up for a single event. A referee will often set a maximum number of players for an event. Otherwise, you run out of little car and Scottie dog pieces fairly quickly.

To manage this, we need to keep track of the maximum number of attendees allowed for each event. We can do that by adding a MaxPlayers field. Also, we'll need to check that we don't go over our maximum. In a conventional database system, you'd count how many players there are by performing an SQL SELECT statement using the COUNT function, then pulling this amount out of the results of the ADO.NET query.

However, in DeKlarit, we can do what we'd like to do as application programmers – we can just add a NumPlayers field and use that to report how many rows there are in the Attendees table for an event. We do that using a calculated field, as shown in Figure 15:



**Figure 15 - Adding a calculated field**

An attribute in a structure can be either a database field (as we've been using so far) or a value calculated from other fields in the structure. These calculated fields are called formulas. NumPlayers has a formula defined as:

```
COUNT( PlayerName )
```

This is an example of a Vertical Formula, which is one that deals with data across a set of rows. The COUNT function returns the number of PlayerNames in the current ConventionEvent. It's also useful to have Horizontal formulas, which operates on data within a single record. In our ConventionEvent structure, suppose we wanted a field that contained the number of seats left. We'd define the formula as:

```
MaxPlayers - NumPlayers
```

Formulas don't care what order they're defined in. If you've got more than one computed field in a structure, DeKlarit will automatically figure out the dependencies and get everything working properly.

Of course, just because we have a MaxPlayers field and a NumPlayers field doesn't mean that DeKlarit will automatically check it. There's still some labor involved... but not much. DeKlarit lets us express constraints using computed values and rules. DeKlarit rules are a set of statements of the form:

```
<action> if <condition>;
```

that are attached to each DeKlarit structure. The language used is a fairly simple one defined by DeKlarit, rather than C# or VB.NET. Rules are declarative rather than procedural - they specify what the rule is, not how to determine the answer. And the order of rules is irrelevant; DeKlarit will figure out what order to apply them automatically based on data dependencies.

For our first rule, let's see how to enforce the MaxPlayers limitation. The rule is:

```
error( EventIsFullException, "Event Is Full" ) if NumPlayers > MaxPlayers;
```

The action here is "error", which will, at runtime, throw an exception object of the specified type if the rule is violated (DeKlarit automatically generates the new exception class as a nested class inside the DataAdapter class). The exception is triggered when the NumPlayers field is greater than MaxPlayers. You can do other things with rules, such as assign default values to fields that are null in the database, add and subtract fields and constants, assign to fields, etc. See the DeKlarit documentation for more details.

## What's the Catch?

DeKlarit makes life easier for those of us who want to concentrate on the application, but still need to keep data in the database. But, as usual with any translation tool, there are some areas that DeKlarit won't help you with.

DeKlarit doesn't give you much control over the generated code or SQL statements. You're limited to defining the input structures and DeKlarit takes it from there. Also, editing the generated code isn't an option as the code will be rewritten the next time you change the structures.

One final gotcha hit us as we were experimenting with the event site. The current version of DeKlarit is SQL Server specific. We tried to run it against an Access database but had no luck. DeKlarit does include a reverse engineering tool that will read in other database formats, but the only database the generated code will talk to is SQL Server.

## OO/RDBMS Nirvana, or at least Co-existence

The convention web site isn't done yet, but Chris is a lot further along than he expected to be after only a couple of days of experimentation. DeKlarit's ability to quickly take my object-oriented design and build a correct relational model from that design is incredibly productive. Being able to code against the OO design rather than having to worry about the needs of database normalization made development on the system much more productive, as the code didn't have to contort itself away from the original design to keep the database happy. And DeKlarit's ability to migrate data between schemas as the model is refined is nothing short of amazing. This capability saved a tremendous amount of time and allowed for experiments that would have been much more time consuming if the data had had to be migrated by hand. For those that, like Chris, don't have a lot of database experience, or even those, like Chris, where the actual representation in the database often takes a backseat to the application itself, DeKlarit has a lot to offer.

DeKlarit is not just the same-old UI-based object-relational mapping software. It doesn't start with the database schema and generate bad objects, nor does it start with the objects and generate bad databases. Instead, DeKlarit starts in the middle, using solid theory to map structure definitions to both SQL and .NET types. The DeKlarit engineers have done a lot of work to provide the ease-of-use of an object-oriented-like development environment, with hard constraints on how the data must be represented in the database to maintain integrity. DeKlarit provides a very real alternative to those of us with the luxury of starting from the application, instead of the database, first.