

RPG – Programme für alle Plattformen

## RPG wird plattformunabhängig

Softwarehäuser suchen nach Möglichkeiten ihre bestehenden Anwendungen auf andere Plattformen zu portieren da Sie vom Markt unter Druck geraten. iSeries-Anwender wünschen sich eine bessere Integration ihrer Programme auf Windows und direkten Zugriff auf Datenbanken wie SQL-Server, Access und Oracle. Wie sich das alles in ASNA VisualRPG.NET umsetzen lässt zeigt dieser Artikel.

### Die Evolution geht weiter

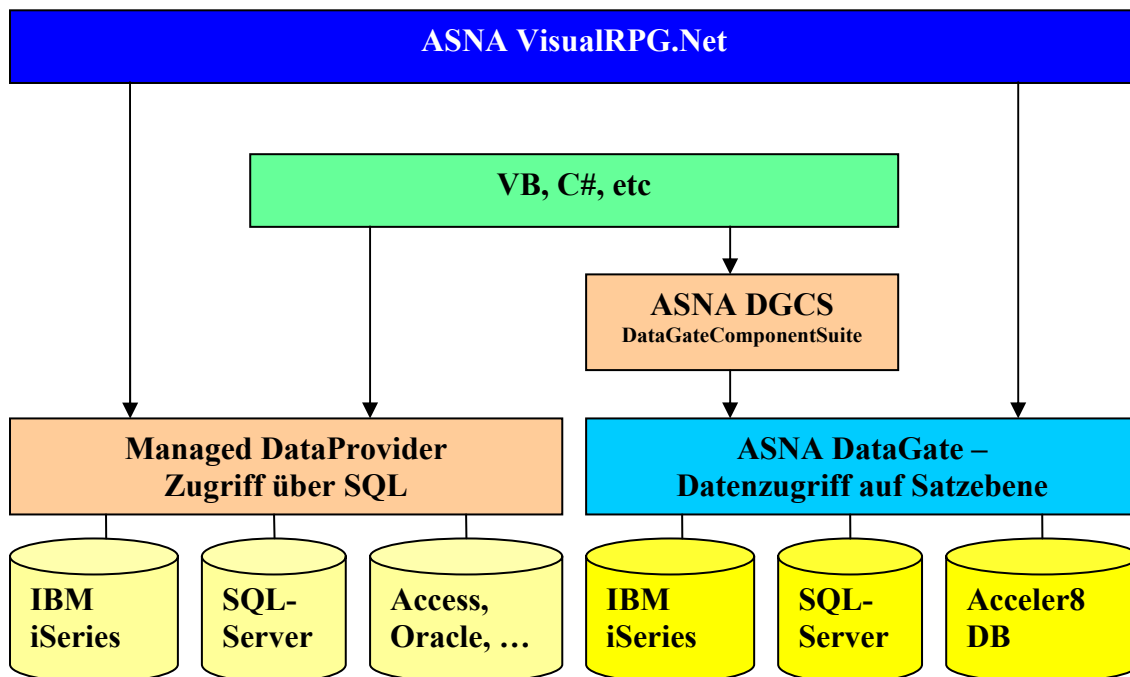
Totgesagte leben bekanntlich länger. Auch für RPG trifft diese Regel zu. Obwohl es sich bei VisualRPG.NET nicht um eine Weiterentwicklung von IBM sondern von ASNA handelt wird VisualRPG.NET von IBM auf der iSeries-Roadmap als Produkt empfohlen. Mit diesem RPG-Compiler und dem Migrationstool MONARCH können iSeries-Programme auf DotNet arbeiten. Dieses Thema hat viele interessante Aspekte, wir widmen uns in diesem Artikel der Plattformunabhängigkeit.

In unserem kleinen Beispielprogramm geht es um eine Adresstabelle die auf 4 Datenbanken liegt auf die wir aus unserem RPG-Programm zugreifen. Es geht um die Datenbanken:

- DB2/iSeries
- Microsoft SQL-Server
- Acceler8DB von ASNA
- Microsoft ACCESS

Dem Datenzugriff unter DotNet liegt das Konzept der ManagedProvider zu Grunde. Es handelt sich dabei um Datenbank-Driver die in DotNet eingebettet sind und die Datenbank optimal betreiben da sie speziell für EINE Datenbank geschrieben wurden. In DotNet kann man beliebig viele ManagedProvider integrieren, daher ist man für alle Plattformen offen und hochperformant da sich sozusagen um jede Datenbank ein ‚Spezialist‘ kümmert.

Auch für den für RPG typischen Zugriff auf Satzebene kümmert sich ein Spezialist – DataGate. DataGate gibt es für die iSeries, den SQL-Server und die Acceler8-DB von ASNA.



Also kann aus einem RPG-Programm auf Satzebene auf diese 3 Datenbanken zugegriffen werden ohne dafür eigens Code zu schreiben.

Auf die Access-Datenbank greifen wir über den ManagedProvider zu. Dazu schreiben wir eine Routine mit Embedded-SQL.

Um das Thema spannend zu machen greifen wir alternativ auf Access so zu dass die Logik des Programms gar nicht mitbekommt dass es sich nicht um eine iSeries-Datenbank handelt auf der die Daten liegen.

So sieht das Programm aus, es hat 5 Buttons von denen jeder einen Datenzugriff auf ‚seiner‘ Datenbank ausführt.

Button-Bezeichnung	Funktion
Chain iSeries	Zugriff auf iSeries
Chain Local	Zugriff auf lokale Datenbank am Entwicklungsrechner
Chain SQL-Server	Zugriff auf Microsoft-SQL-Server
SQL-Access	SQL-Zugriff auf die Access-Datenbank
SQL-Chain-Access	SQL-Zugriff auf Access-Datenbank der sich für die Logik wie ein Zugriff mit CHAIN darstellt.

Dieser Name SQL-Chain – Access schreit regelrecht nach Verbesserungsvorschlägen, bitte senden an [c.neissl@niceware.at](mailto:c.neissl@niceware.at)

Der Sinn ist einen Zugriff auf die Datenbank so zu implementieren dass es für die übergeordnete Logik keinen Unterschied macht welche Datenbank darunterliegt. Da wird schon deutlicher wo die Reise hingeht.

Wir wollen Software auf den Markt bringen, aber heute akzeptiert kaum mehr ein Unternehmen dass die Software von der Hardware abhängig ist. Größere Unternehmen sind da offener aber wir wollen unsere Software auch kleineren Unternehmen verkaufen die bereits Serversysteme einsetzen und nicht bereit sind dafür extra Hardware anzuschaffen und zu betreuen.

### Und so funktioniert's mit RPG

Der Chain wird in einer einzigen Routine ausgeführt. Diese Routine bekommt die Datenbank-Verbindung übergeben, eröffnet die Datei und liest mit CHAIN die Daten ein:

```
Chain Customer_num Key( kyCMCustNo ) Err(*extended)
if %found()
  // Anzeigefelder aus Datenfelder füllen
  ExSR FillFromFields
  stb.Text = "Chain auf " + DBName.DBName.ToString() + "..."
endif
```

Diese Funktion wird aus den obersten 3 Buttons aufgerufen, jeweils mit der eigenen Datenbank-Verbindung die im Programm deklariert wird:

```
DclDB iSeriesDB      DBName( "ATNICE" )
DclDB ProdDB        DBName( "DG NET Local" )
DclDB SQLDB         DBName( "ASNA SQL DB" )
```

Und so sehen die Aufrufe für das CHAIN aus:

```
// aus iSeries lesen
ChainiSeries(txtCMCustNo.text, iSeriesDB)

...

// aus lokaler DB lesen
ChainiSeries(txtCMCustNo.text, ProdDB)

...

// aus SQL-Server lesen
ChainiSeries(txtCMCustNo.text, SQLDB)
```

Der Zugriff auf Access erfolgt über Embedded-SQL, das heißt in DotNet – ADO-DotNet.

```
daConn = *new OleDbDataAdapter( +
  "SELECT * FROM CustMastNew WHERE CMCustNo = " +
  %trim(KundenNR) + ";", dbConn )
daConn.Fill(dsRead, "CustMastNew")
```

Und nun werden einfach die Datenfelder der iSeries mit dem Rückgabewert des SQL befüllt.

### **Simpel und doch einmalig**

Logisch was sollte man auch sonst mit den Datenfeldern machen, was ist da so großartig ?

Bei genauerem Hinsehen wird man feststellen dass dieses RPG-Programm auch unabhängig von der iSeries laufen kann. Wie kommt es dann zu iSeries Datenfeldern, wieso sind die in einem Programm das auf der Datenbank Access, MySQL, ORACLE, etc. läuft vorhanden ?

Der Clou ist dass der Compiler zur Umwandlungszeit die Datenfelder im Programm bereitstellt. Wie damit zur Laufzeit umgegangen wird und woher sie befüllt werden ist eine andere Geschichte.

Diese Funktion kann dazu benutzt werden RPG-Programme mit der iSeries zu entwickeln und auch ohne iSeries und DataGate, mit einer alternativen Datenbank zu betreiben.

Alle die das probieren möchten können unter meiner eMail-Adresse [c.neissl@niceware.at](mailto:c.neissl@niceware.at) ein komplettes Paket mit VisualStudio, VisualRPG.NET und 200 Seiten Leitfaden in deutscher Sprache gratis bestellen. Freue mich auf die Diskussionen.