Dependencies and References in SQL Server

red-gate.com/simple-talk/sql/t-sql-programming/dependencies-and-references-in-sql-server

September 24, 2015

Phil Factor
24 September 2015

39

It is important for developers and DBAs to be able to determine the interdependencies of any database object. Perhaps you need to work out what process is accessing that view you want to alter, or maybe find out whether that table-type you wish to change is being used. What are all these dependencies? How do you work out which are relevant? Phil Factor explains.

Dependencies and References in SQL Server.

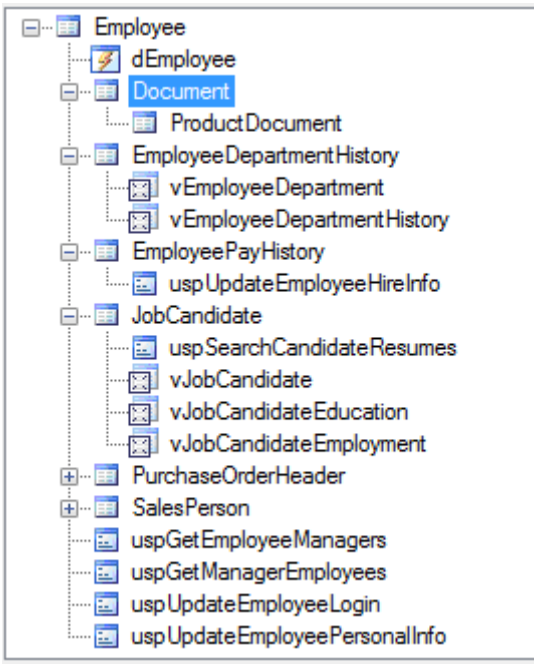
In a relational database, it isn't just the data that is related, but the database objects themselves. A view, for example, that references tables is dependent upon them, and wherever that view is used the function, procedure or view that uses it depends on it. Those tables referred to by the view may in turn contain user-defined types, or could be referenced to other tables via constraints. By its very nature, any SQL Server database will contain a network of inter-dependencies.

SQL Server objects, such as tables, routines and views, often depend on other objects, either because they refer to them in SQL Expressions, have constraints that access them, or use them. There may be other objects that are, in turn, dependent on them. Dependencies grow like nets. It isn't just foreign keys or SQL references that cause dependencies, but a whole range of objects such as triggers, user-defined types and rules. It can complicate any changes to a database by requiring a specific order of operations within a database build script, or migration script. If you get it wrong, you'll get a whole range of errors like "Cannot drop xxx 'MyName' because it is being referenced by object 'HisName' There may be other objects that reference this yyy". or " xxx 'MyName' references invalid xxx 'HerName'." Basically, objects need to be deleted or altered in a particular order. In a well-designed SQL Server database, or set of linked databases, it is easy to determine these dependencies, and work out the right sequence for doing things.

Finding dependencies via SSMS

Most of us need to think very little about finding out about dependencies, since SMO allows SSMS to get dependency information for us for any list of database objects, and display it in a tree structure. If you wish to know what dependencies an object has, or what it in turn depends on, You just right-click the object in the object explorer pane and click on 'view dependencies' in the context menu that then appears.

Your re-engineering work must take these dependencies into account. If you need to, for example, delete a column, your work must start at the 'leaves' to make sure that nothing untoward references that column. If you change a user-defined table type, then you need to check wherever it has been used in the database. SSMS uses SMO to get this information. You can get the same information yourself, if you need to, by using a PowerShell script to get the same information from SMO. Accessing, in this example, the same table from the same database …




```
Import-Module sqlserver -DisableNameChecking
#load the SQLPS functionality for getting the registered servers

#-- just to save typing ----
$MS = 'Microsoft.SQLServer'
$My = "$MS.Management.Smo"
$Mc = "$MS.Management.Common"

$dbname = "Adventureworks2016" #the database we want
$tableName = 'Employee' #the table to investigate
$schemaname = 'HumanResources' #the schema that the table is in
$serverName = "Philf01" #the name of the server
$credentials = 'user=Philfactor;pwd=ismellofpoo4U'
# if SQL Server credentials, use 'user id="MyID;password=MyPassword"'
$DoWeLookForParentObjects = $false
#look for the parent objects for each object-set to false for child objects

#create the connection string
$connectionString = `
"Data
Source=$serverName;$credentials;pooling=False;multipleactiveresultsets=False;"
#connect to the server
try
{
  $sqlConnection = `
    new-object System.Data.SqlClient.SqlConnection $connectionString
  $conn = new-object "$Mc.ServerConnection" $sqlConnection
  $srv = new-object Microsoft.SqlServer.Management.Smo.Server $conn
}
catch
{
  "Could not connect to SQL Server instance '$_.servername':
$($error[0].ToString() +
    $error[0].InvocationInfo.PositionMessage). Script is aborted"
  exit -1
```

```
}
#now get the SMO object for the table that we want.
$success = $false
if ($srv.Databases[$dbname] -ne $null)
{
  $db = $srv.Databases[$dbname]
  if ($db.Tables.Contains($tableName, $schemaName))
  { $table = new-object "$My.Table" ($db, $tableName, $schemaName);
    $success = $true;}
}
if (-not $success)
{
  Write-error "Couldn't find the $TableName table in the $dbname database"
  exit -1
}
#now  we set up the scripter object
$scr = New-Object "$My.Scripter"
#now choose options for the scripter that we need to get dependency order
$options = New-Object "$My.ScriptingOptions"
$options.DriAll = $True
$options.AllowSystemObjects = $false
$options.WithDependencies = $True
$scr.Options = $options
$scr.Server = $srv

$VerbosePreference = "Continue"
#we set up a URNcollection to contain our objects for analysis
# (only one in this example)
$urnCollection = new-object "$my.UrnCollection"
$urnCollection.Add([Microsoft.SqlServer.Management.Sdk.Sfc.Urn]$table.Urn)
#now we set up an event listnenr go get progress reports
$ProgressReportEventHandler =
[Microsoft.SqlServer.Management.Smo.ProgressReportEventHandler] `
  { Write-Verbose "analysed '$($_.Current.GetAttribute('Name'))'" }
$scr.add_DiscoveryProgress($ProgressReportEventHandler)
#create the dependency tree
$dependencyTree = `
  $scr.DiscoverDependencies($urnCollection, $DoWeLookForParentObjects)
#look for the parent objects for each object
#and walk the dependencies to get the dependency tree.
$depCollection = $scr.WalkDependencies($dependencyTree);
#now we just show the dependency tree!
$depCollection | %{
  "$(if ($_.IsRootNode -eq $true) { 'root: ' }
    ) $($_.Urn.GetAttribute('Schema', $_.Urn.Type)
    ).$($_.Urn.GetAttribute('Name', $_.Urn.Type)
    )--$($_.Urn.Type)"
}

$VerbosePreference = "SilentlyContinue"
```

Is there another way of doing this, hopefully in SQL? Well, yes there is, but I'll be showing how to do that at the end of the article, and providing the code. But firstly, I'll need to explain where some of the complications are.

Soft and hard Dependencies

Dependencies are of two types. There are 'soft' dependencies; references to other objects in SQL code that are exposed by sys.sql_expression_dependencies, and 'hard' dependencies that are exposed by the object catalog views. 'Hard' dependencies are inherent in the structure of the database, whereas code can reference objects in another database on the same server or on another server.

Soft Dependencies

These soft dependencies are all recorded and are available from the `sys.sql_expression_dependencies`. If someone has allowed ad-hoc SQL to be generated by applications rather than use stored procedures or functions, you are free to weep at this point, because you have missed out on getting all this essential information and will find it very hard to refactor your database.

'Soft' Dependencies happen when you have a routine (that is a procedure, function rule, constraint or anything else with code in it) that refers to another entity, possibly in another database. By dint of a SQL Expression in, say, a View, you can make that view dependent on one or more other objects. This dependency information is maintained by the database, but not for rules, defaults, temporary tables, temporary stored procedures, or system objects, and only when the referenced entity appears by name in a persisted SQL expression of the referencing entity.

There are two types of soft dependency

- **Schema-bound dependency**
  This is a relationship between two entities that means that there is an error if there is an attempt to drop the referenced entity when the referencing entity exists. This happens when a view or user-defined function uses the `WITH SCHEMABINDING` clause, or when a table has a `CHECK` or `DEFAULT` constraint or a computed column that references a user-defined function, user-defined type, or XML schema collection. If you execute an **ALTER TABLE** statement on a table that are referenced by views or UDFs that have schema binding, then you will get an error if the statement affects the view definition. The `WITH SCHEMABINDING` clause binds the view or UDF to the schema of the underlying base tables that they reference so that they cannot be modified in a way that would affect the view definition. The view or UDF must be dropped first. .
- **Non-schema-bound dependency**
  This is a dependency relationship between two entities that does not trigger an error when the referenced entity is dropped or modified.

Here is a simple query to find out in AdventureWorks, all the references that `Sales.vIndividualCustomer` makes

```
SELECT Coalesce(Object_Schema_Name(referencing_id) + '.', '')
      + --likely schema name
  Object_Name(referencing_id) + --definite entity name
  Coalesce('.' + Col_Name(referencing_id, referencing_minor_id), '') AS
referencing,
  Coalesce(referenced_server_name + '.', '')
  + --possible server name if cross-server
  Coalesce(referenced_database_name + '.', '')
  + --possible database name if cross-database
  Coalesce(referenced_schema_name + '.', '')
  + --likely schema name
  Coalesce(referenced_entity_name, '')
  + --very likely entity name
  Coalesce('.' + Col_Name(referenced_id, referenced_minor_id), '') AS referenced
  FROM sys.sql_expression_dependencies
  WHERE referencing_id = Object_Id('Sales.vIndividualCustomer')
  ORDER BY referenced;
```

And here is a query that finds out all the objects that reference **'Sales.SalesOrderHeader'**

```
SELECT Coalesce(referenced_server_name + '.', '')
      + --possible server name if cross-server
  Coalesce(referenced_database_name + '.', '')
      + --possible database name if cross-database
  Coalesce(referenced_schema_name + '.', '') + --likely schema name
  Coalesce(referenced_entity_name, '') + --very likely entity name
  Coalesce('.' + Col_Name(referenced_id, referenced_minor_id), '') AS referencing,
  Coalesce(Object_Schema_Name(referencing_id) + '.', '') + --likely schema name
  Object_Name(referencing_id)
  + --definite entity name
  Coalesce('.' + Col_Name(referencing_id, referencing_minor_id), '') AS referenced
  FROM sys.sql_expression_dependencies
  WHERE referenced_id = Object_Id('Sales.SalesOrderHeader')
  ORDER BY referenced;
```

`Sys.sql_expression_dependencies` also has the information as to whether the dependency is schema-bound or not.

Here is a routine that shows you the soft dependency order of the objects in your database, and lists the external dependencies of any objects. (note that a lot of entities in a database aren't classed as objects.)

```
CREATE FUNCTION dbo.DependencyOrder ()
/*
summary:   >
This table-valued function is designed to give you the order in which
database objects should be created in order for a build to succeed
without errors. It uses the sys.sql_expression_dependencies table
for the information on this.
it actually only gives the level 1,,n so within the level the order
is irrelevant so could, i suppose be done in parallel!
```

```
It works by putting in successive passes, on each pass adding in objects
who, if they refer to objects, only refer to those already in the table
or whose parent object is already in the table. It goes on until no more
objects can be added or it has run out of breath. If it does more than
ten iterations it gives up because there must be a circular reference
(I think that's impossible)
Revisions:
- Author: Phil Factor
   Version: 1.0
   Modification: First cut
   date: 3rd Sept 2015
example:
     - code: Select * from dbo.DependencyOrder() order by theorder desc
returns:   >
a table, giving the order in which database objects must be built
*/
RETURNS @DependencyOrder TABLE
  (
  TheSchema VARCHAR(120) NULL,
  TheName VARCHAR(120) NOT NULL,
  Object_id INT PRIMARY KEY,
  TheOrder INT NOT NULL,
  iterations INT NULL,
  ExternalDependency VARCHAR(2000) NULL
  )
AS
  -- body of the function
  BEGIN
    DECLARE @ii INT, @EndlessLoop INT, @Rowcount INT;
    SELECT @ii = 1, @EndlessLoop = 10, @Rowcount = 1;
    WHILE @Rowcount > 0 AND @EndlessLoop > 0
      BEGIN
        ;WITH candidates (object_ID, Parent_object_id)
         AS (SELECT sys.objects.object_id, sys.objects.parent_object_id
               FROM sys.objects
                 LEFT OUTER JOIN @DependencyOrder AS Dep
                 --not in the dependency table already
                   ON Dep.Object_id = objects.object_id
               WHERE Dep.Object_id IS NULL AND type NOT IN ('s', 'sq', 'it'))
        INSERT INTO @DependencyOrder (TheSchema, TheName, Object_id,
TheOrder)
        SELECT Object_Schema_Name(c.object_ID), Object_Name(c.object_ID),
          c.object_ID, @ii
          FROM candidates AS c
            INNER JOIN @DependencyOrder AS parent
              ON c.Parent_object_id = parent.Object_id
        UNION
        SELECT Object_Schema_Name(object_ID), Object_Name(object_ID),
          object_ID, @ii
          FROM candidates AS c
          WHERE Parent_object_id = 0
            AND object_ID NOT IN
                (
                SELECT c.object_ID
                  FROM candidates AS c
                    INNER JOIN sys.sql_expression_dependencies
```

```
                    ON Object_id = referencing_id
                  LEFT OUTER JOIN @DependencyOrder AS ReferedTo
                    ON ReferedTo.Object_id = referenced_id
                 WHERE ReferedTo.Object_id IS NULL
                  AND referenced_id IS NOT NULL
                  --not a cross-database dependency
               );
      SET @Rowcount = @@RowCount;
      SELECT @ii = @ii + 1, @EndlessLoop = @EndlessLoop - 1;
    END;
  UPDATE @DependencyOrder SET iterations = @ii - 1;
  UPDATE @DependencyOrder
    SET ExternalDependency = ListOfDependencies
    FROM
      (
      SELECT Object_id,
        Stuff(
            (
            SELECT ', ' + Coalesce(referenced_server_name + '.', '')
                + Coalesce(referenced_database_name + '.', '')
                + Coalesce(referenced_schema_name + '.', '')
                + referenced_entity_name
              FROM sys.sql_expression_dependencies AS sed
              WHERE sed.referencing_id = externalRefs.object_ID
                AND referenced_database_name IS NOT NULL
                AND is_ambiguous = 0
            FOR XML PATH(''), ROOT('i'), TYPE
            ).value('/i[1]', 'varchar(max)'),1,2,'' )
              AS ListOfDependencies
        FROM @DependencyOrder AS externalRefs
      ) AS f
      INNER JOIN @DependencyOrder AS d
        ON f.Object_id = d.Object_id;

  RETURN;
 END;
GO
```

there are also two functions that provide information on soft dependencies

- The `sys.dm_sql_referenced_entities` Dynamic Management Function (DMF) returns every user-defined entity that is referenced by name in the definition of the referencing database object that you specify.
- The `sys.dm_sql_referencing_entities` DMF returns every user-defined entity in the current database that references the user-defined object, type (alias or CLR UDT), XML schema collection, or partition function that you specify.

Hard Dependencies

'Hard' dependencies can happen whenever an object can reference another one. The rules are complicated.

SQL Server has a number of types of objects and a whole lot of other entities that aren't classed as database objects. The rules of what can reference what is best expressed as a table

Referencer	Can refer to ('reference')											
	Tables	UDTTs	UDTs	UDF	Procedure	Triggers	Defaults	Rules	XMLSchemaCollections	Assembly	Partition Scheme	Partition functions
Table	✓		✓			✓	✓	✓	✓		✓	
Table Types		✓	✓						✓	✓		
Types							✓	✓				
Procedure									✓	✓		
UDF										✓		
Assembly			✓							✓		
View	✓		✓	✓		✓					✓	
Partition Scheme												✓
Plan Guide				✓	✓							
Synonym	✓			✓	✓	✓	✓	✓				
Sequence		✓										

Dependencies and Build Scripts.

Databases, in general, have to be built in the right order. This order avoids building anything that relies on an object that hasn't been built yet. An easy way of doing this is to create objects in a particular order of object types. The downside of doing this is that objects that should really go together for clarity when inspecting scripts, such as tables, constraints, extended properties and indexes, get scattered in to different places for the convenience of an easy compilation. Clarity is sacrificed for convenience: also you will still need to do certain routines in soft dependency order.

An exception to this is the CREATE SCHEMA statement that allows its contents to be created by CREATE SCHEMA in any order within the subsequent list, except for views that reference other views. In that case, the referenced view must be created before the view that references it. It is actually possible to use the CREATE SCHEMA statement without the schema name, but still allow the build list to be specified in any order other than views that reference views. However, this special syntax is deprecated.

SMO likes to do build scripts in ObjectType order in a build script. The script starts with Database properties, followed by Schemas, XML Schema Collections and Types: none of which can have dependent objects. Table Types and Procedures come next. Then, in dependency order, Functions, Tables and Views. Then come Clustered indexes, non-clustered indexes, Primary XML Indexes, XML indexes, Default Constraints, Foreign keys Check constraints, triggers and lastly, extended properties. This order minimises the

shuffling that needs to be done. Stored procedures are unique amongst modules or routines in that they have deferred compilation, which neatly kicks soft dependencies into touch for builds.

Cross-server, cross-database and cross-schema dependencies

Cross-database dependencies

'Soft' dependencies are likely to refer to objects in other databases. These can be on the same server or on a different server. These can both be obtained from sys.sql_expression_dependencies. Sometimes, these can cause difficulties in the delivery process because they aren't properly encapsulated in an interface of some sort, and aren't wired into the build process. Often, these external references need to be 'mocked' in development and only assigned to their destination during test or staging. This means that the actual routine that makes the external reference must be related to the particular delivery environment (e.g. Integration Test, UAT, Performance Testing, staging and production), and the development build will have the source of the 'mock' only. Each delivery environment is assigned the correct version of the code. The sys.sql_expression_dependencies is your friend in ensuring that all these external dependencies are tracked, and that none slip through the net to cause build problems. A warning though: XML documents are considered by SQL Server to be external databases and produce false positives when attempting to identify cross-database dependencies.

Cross-schema dependencies

I have worked with database developers who maintain hand-cut database build scripts that are done in a way that preserve dependency order whilst aiming at clarity. It is a pleasure to inspect, when done by one of the more professional developers, since it is generally well-documented. These are generally done, and saved in source-control, at schema level to allow more than one developer to work on the database concurrently. Cross-schema references are relevant here because the best practice is to reduce these to a minimum to allow as much autonomy as possible to the individual developer, avoid merges, and have as few build-breakages as possible. Here, with cross-schema references, both soft and hard dependencies are possible. Schema builds, unlike database builds, can list their object creation scripts in almost any order after the CREATE SCHEMA without errors.

Walking particular dependency types.

The reality of many dependency-based operations is that only one type of dependency is relevant, and not even the individual dependency chains. It just depends on 'layers'. Take tables, for example. If you had a list in which the tables of a database were layered according to the fact that all their dependencies were satisfied by the preceding layer or below, then, as long as you do the operation to all of the layer below before the current one, then you aren't going to break a dependency. I use this type of routine to to do fast-

BCP loads into tables as part of a build, but it is also useful to establish an order of build if your individual table scripts contain embedded foreign key definitions as either column or table constraint definitions.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
IF OBJECT_ID (N'TempDB..#TablesInDependencyOrder') IS NOT NULL
  DROP PROCEDURE #TablesInDependencyOrder
GO
Create PROCEDURE #TablesInDependencyOrder
/**
summary:
  For the table(s) you specify, this routine returns a table containing all the related
tables
  in the current database, their schema, object_ID, and their
  dependency level.
  You would use this for deleting the data from tables or BCPing in the data.
Author: Phil Factor
Revision: 1.0 First cut
Created: 25th september 2015
example:
    -
      Declare @tables Table( TheObject_ID INT NOT null,
      TheName SYSNAME NOT null,TheSchema SYSNAME NOT null,
      HasIdentityColumn INT NOT null,TheOrder INT NOT null)
      insert into @tables
        Execute #TablesInDependencyOrder
      Select * from @Tables
returns:
       TheObject_ID INT,--the tables' object ID
       TheName SYSNAME, --the name of the table
       TheSchema SYSNAME, --the schema where it lives
       TheOrder INT) --Order by this column
**/
AS
SET NOCOUNT ON;
DECLARE @Rowcount INT, @ii INT
CREATE TABLE #tables (
  TheObject_ID INT,--the tables' object ID
  TheName SYSNAME, --the name of the table
  TheSchema SYSNAME, --the schema where it lives
  TheOrder INT DEFAULT 0) --we update this later to impose an order
/* We'll use a SQL 'set-based'  form of the topological sort. Firstly
we will read in all the desired tables identifying
the start nodes as level 1 These "start nodes" have no incoming edges
at least one such node must exist in an acyclic graph*/
INSERT  INTO #tables (Theobject_ID, TheName, TheSchema, TheOrder)
  SELECT  DISTINCT
      TheTable.OBJECT_ID, TheTable.NAME,
      object_schema_name(TheTable.OBJECT_ID) AS [Schema],
      CASE WHEN --referenced.parent_object_ID IS NULL AND
            referencing.parent_object_ID IS NULL THEN 1 ELSE 0 END AS
TheOrder
```

```
    FROM  sys.tables TheTable
    -- LEFT OUTER JOIN sys.foreign_Keys referenced
    -- ON referenced.referenced_Object_ID = TheTable.object_ID
    LEFT OUTER JOIN sys.foreign_Keys referencing
     ON referencing.parent_Object_ID = TheTable.object_ID
SEIECT @Rowcount=100,@ii=2
--and then do tables successively as they become 'safe'
WHILE @Rowcount > 0
  BEGIN
  UPDATE  #tables
  SET   TheOrder = @ii
  WHERE   #tables.TheObject_ID IN (
      SELECT  parent.TheObject_ID
      FROM  #tables parent
          INNER JOIN sys.foreign_Keys
            ON sys.foreign_Keys.parent_Object_ID = parent.Theobject_ID
          INNER JOIN #tables referenced
            ON sys.foreign_Keys.referenced_Object_ID = referenced.Theobject_ID
           AND sys.foreign_Keys.referenced_Object_ID <> parent.Theobject_ID
      WHERE   parent.TheOrder = 0--i.e. it hasn't been ordered yet
      GROUP BY parent.TheObject_ID
      HAVING  SUM(CASE WHEN referenced.TheOrder = 0 THEN -20000
              ELSE referenced.TheOrder
            END) > 0--where all its referenced tables have been ordered
  )
  SET @Rowcount = @@Rowcount
  SET @ii = @ii + 1
  IF @ii > 100
    BREAK
END
SELECT TheObject_ID,TheName,TheSchema,TheOrder
FROM #tables order by TheOrder
IF @ii > 100 --not a directed acyclic graph (DAG).
  RAISERROR ('Cannot load in tables with mutual references in foreign keys',16,1)
IF EXISTS ( SELECT  * FROM #tables WHERE TheOrder = 0 )
  RAISERROR ('could not do the topological sort',16,1)
GO
```

This sort of technique only works with some operations. With others, you need to follow a dependency branch from a particular object to track all the objects that a particular object depends on, and what depends on the object. This requires a more surgical approach based on the dependency tracker in SSMS. For a broader perspective that allows you to inspect an entire database, as well as to zoom in on detail, then SQL Dependency Tracker is ideal.

It_Depends

So is there another way to just simply list the dependencies, in other words the entities that depend on an object, and the ones that the object depends on, other than using PowerShell or the dependency displayer within SSMS? I use my own SQL-Based home-brewed dependency tracker for the work I need it for. It is in SQL but its code is a bit long to list here in the article. It can be viewed here. It shows a lot of what I've described in this article and in more detail. It gives you a similar display to the one in SSMS, but you can use it for other purposes as well, and it is rather faster! You can download it from the head of the article.

You use it like this ...

```
1  Use AdventureWorks
2  SELECT  space(iteration * 4) + TheFullEntityName + ' (' + rtrim(TheType) + ')'
3  FROM    dbo.It_Depends('Employee',0)
4  ORDER BY ThePath
```

...to give a hierarchy like this.

```
HumanResources.Employee (U)
    dbo.ufnGetContactInformation (U)
    dbo.uspGetEmployeeManagers (U)
    dbo.uspGetManagerEmployees (U)
    HumanResources.dEmployee (U)
    HumanResources.EmployeeAddress (U)
    HumanResources.EmployeeDepartmentHistory (U)
    HumanResources.EmployeePayHistory (U)
    HumanResources.JobCandidate (U)
        HumanResources.vJobCandidate (U)
        HumanResources.vJobCandidateEducation (U)
        HumanResources.vJobCandidateEmployment (U)
    HumanResources.uspUpdateEmployeeHireInfo (U)
    HumanResources.uspUpdateEmployeeLogin (U)
    HumanResources.uspUpdateEmployeePersonalInfo (U)
    HumanResources.vEmployee (U)
    HumanResources.vEmployeeDepartment (U)
    HumanResources.vEmployeeDepartmentHistory (U)
    Purchasing.PurchaseOrderHeader (U)
        Purchasing.iPurchaseOrderDetail (U)
        Purchasing.PurchaseOrderDetail (U)
        Purchasing.uPurchaseOrderDetail (U)
        Purchasing.uPurchaseOrderHeader (U)
    Sales.SalesPerson (U)
        Sales.SalesOrderHeader (U)
            Sales.CalculateSalesOrderTotal (U)
            Sales.iduSalesOrderDetail (U)
            Sales.SalesOrderDetail (U)
                Sales.OrderWeight (U)
            Sales.SalesOrderHeaderAudit (U)
            Sales.SalesOrderHeaderSalesReason (U)
        Sales.SalesPersonQuotaHistory (U)
        Sales.SalesTerritoryHistory (U)
        Sales.Store (U)
            Sales.iStore (U)
            Sales.iuIndividual (U)
            Sales.StoreContact (U)
            Sales.vStoreWithDemographics (U)
        Sales.uSalesOrderHeader (U)
    Sales.vSalesPerson (U)
    Sales.vSalesPersonSalesByFiscalYears (U)
```

It is a bit rugged when compared with what you can achieve via SSMS, but it is quicker, and great for SQL development work when you are having to check out a rats-nest of dependencies. (To use it with SQL Server 2008, you'll need to nick out the statement that accesses `sys.sequences`, together with it's accompanying `UNION ALL)`

Conclusions

If you can be sure about the way that the database objects you're working on depend on each other and upon other database objects, both in the database and outside it, then it becomes a lot easier and more restful to re-engineer a database. Refactoring becomes less like an extreme sport, and more like knitting. If your database sticks to the convention of

using only compiled routines such as stored procedures and functions, then you will know what references that table you want to get rid of, or what needs to be reworked when you alter that user-defined table type. Any tool, or combination of tools, that track dependencies are going to be very useful to you.